

AN MBONE RECORDER/PLAYER

A thesis submitted to the faculty of
San Francisco State University
in partial fulfillment of the
requirements for the
degree

Master of Science
in
Computer Science

by

Gary Hoo

San Francisco, California

December, 1997

Copyright by
Gary Hoo
1997

CERTIFICATION OF APPROVAL

I certify that I have read *An MBone Recorder/Player* by Gary Hoo, and that in my opinion this work meets the criteria for approving a thesis submitted in partial fulfillment of the requirements for the degree: Master of Science in Computer Science at San Francisco State University.

Jozo Dujmovic
Professor of Computer Science

Marguerite Murphy
Professor of Computer Science

William Johnston
Lecturer of Computer Science

AN MBONE RECORDER/PLAYER

Gary Hoo
San Francisco State University
1997

In the last five years, researchers have developed and deployed protocols and applications that allow videoconferencing over the Internet using IP multicast. The portions of the Internet that can communicate via IP multicast are known collectively as the Multicast Backbone, or MBone. The MBone is now routinely used for public lectures as well as private meetings.

As the MBone has grown, the number of available offerings (conferences, public and private) has grown, and a need for tools to record and to replay MBone conferences has arisen. This paper describes such a tool, the DPSS Multimedia Recorder/Player (DMRP), which uses the Distributed-Parallel Storage System (DPSS) as its storage device.

I certify that the Abstract is a correct representation of the content of this thesis.

Thesis Committee Chair

Date

ACKNOWLEDGEMENTS

First and foremost, my thanks to Bill Johnston. This work is the result of the time at LBL that he made possible.

Jason Lee helped in more ways than either of us probably recalls, as a sounding board for ideas, a troubleshooter during development stalls and bug hunts, and even as a voice of sanity during rough times.

Thanks to Keith Beattie, Chris Brooks and John Taylor for suggestions, insights, cautionary tales of technical and bureaucratic snafus, and most of all, many hours of commiseration and encouragement.

The work described in this paper is supported by the U.S. Department of Energy, Office of Energy Research, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division (<http://www.er.doe.gov/production/octr/mics/>) under contract DE-AC03-76SF00098 with the University of California, and by DARPA, Information Systems Technology Office (<http://www.ito.darpa.mil/>). This is document LBNL-41145.

TABLE OF CONTENTS

List of Tables	viii
List of Figures	ix
List of Appendices	x
1 INTRODUCTION	1
2 BACKGROUND	4
2.1 Multicasting	4
2.1.1 Multicasting on the Internet	5
2.1.2 A New Protocol: RTP/RTCP	8
2.1.2.1 Limitations of UDP and TCP	8
2.1.2.2 RTP	11
2.1.2.3 RTCP	16
2.1.2.4 Protocol Customization	24
2.1.3 Other Internet Conferencing Protocols	25
2.1.4 Existing MBone applications	32
2.2 The Distributed-Parallel Storage System	35
2.2.1 Purpose	35
2.2.2 Functionality	36
2.2.3 Architecture and Client Use	38
3 FUNCTIONALITY AND DESIGN ISSUES	42
3.1 RTP/RTCP Packet Management	42
3.2 DPSS Constraints	43
3.3 Other Design Considerations	45
3.4 Design	46
4 IMPLEMENTATION AND ARCHITECTURE OF THE DMRP	49
4.1 Introduction	49
4.1.1 C++ Terminology and Notation	50
4.1.2 C++ Standard Template Library	51
4.1.3 Threads	52
4.2 DMRP Classes	54

4.2.1 Generic Networking Classes	55
4.2.1.1 IPNetAP	55
4.2.1.2 IPAddr	57
4.2.2 RTP/RTCP State Management Classes	57
4.2.2.1 RTPSource	57
4.2.2.2 RTPSourceList	59
4.2.2.3 RTPSession	60
4.2.3 RTP Packet-Handling Classes	61
4.2.3.1 RTPFormat	61
4.2.3.2 Timer	62
4.2.3.3 RTPHandler	63
4.2.3.4 RTPDataHandler	64
4.2.3.5 RTPCtrlHandler	67
4.2.4 RTP/RTCP Protocol Management Class: RTPComm	73
4.2.5 DPSS Interface Class: DPSSPOC	78
4.2.6 DMRP Session Management Class: SessionManager	83
4.2.6.1 Construction and Initialization	83
4.2.6.2 Recording	84
4.2.6.3 Automatic Suspension of Recording	87
4.2.6.4 Playback	88
4.2.6.5 Shutdown	91
4.2.7 Application Status Class: AppStatus	92
4.3 Recorder	95
4.4 Player	102
5 PERFORMANCE AND SCALABILITY	106
6 FUTURE WORK	110
7 CONCLUSIONS	115
REFERENCES	117
APPENDICES	122

LIST OF TABLES

Table	Page
1. RTCP Packet Types	18
2. RTCP SDES Item Types	22

LIST OF FIGURES

Figure	Page
1. RTP Fixed Packet Header	13
2. RTCP Common Packet Header	17
3. RTCP SR Packet	19
4. RTCP SDES Packet	21
5. RTCP SDES Item	22
6. DMRP Packet Manager (conceptual design)	43
7. DMRP design	47
8. DMRP recorder and other MBone tools	96
9. DMRP recorder (detail)	97
10. DMRPSessionInfo structure	98
11. DMRP player and client applications	102
12. DMRP player (detail)	103

LIST OF APPENDICES

Appendix	Page
A Testing Methodologies and Details	122
A.1 DPSS Throughput Test Methodology	122
A.2 DMRP Test Methodology	124
B DMRP Source Code	126
B.1 DMRP Shared Classes	126
B.1.1 Networking Classes	126
B.1.1.1 NetAddr	126
B.1.1.2 IPAddr	127
B.1.1.3 NetAP	128
B.1.1.4 IPNetAP	128
B.1.2 RTP/RTCP Classes	129
B.1.2.1 RTPSource	129
B.1.2.2 RTPSourceList	132
B.1.2.3 RTPSession	132
B.1.2.4 RTPFormat	133
B.1.2.5 Timer	134
B.1.2.6 RTPHandler	136
B.1.2.7 RTPDataHandler	137
B.1.2.8 RTPCtrlHandler	138
B.1.2.9 RTPComm	142
B.1.3 DPSS I/O Classes	145
B.1.3.1 DPSSState	145
B.1.3.2 StorageIOBase	145
B.1.3.3 DPSSPOC	145
B.1.4 SessionManager	147
B.1.5 AppStatus	150
B.2 DMRP common functions	150
B.3 Recorder	159
B.3.1 Common definitions	159
B.3.2 listener.cc	160

B.3.3 listener_utils.cc	170
B.4 Player	175
B.4.1 Common definitions	175
B.4.2 player.cc	176
B.4.3 player_utils.cc	189

CHAPTER I

INTRODUCTION

Videoconferencing—comprising audio, video, and possibly graphical data such as slides—requires timely transmission of potentially large amounts of data to and from multiple sources. One way—the most obvious way—to transfer the data is via a transport infrastructure that provides explicit bandwidth and other service guarantees, such as traditional telephone circuits. In such environments, resources are set aside for the sole use of the conference members and for the duration of the conference, guaranteeing that loss or delay because of bandwidth contention (congestion) will not occur.

The Internet is a poor fit for this model of data transfer. Its main transport protocols, TCP and UDP, provide neither timeliness guarantees nor resource reservation. Congestion, while not inevitable, cannot be eliminated as a possibility by end users because they cannot control intermediate resources, i.e., the nodes that forward the data between the users. And many-to-many communication is expensive in bandwidth terms: it requires transmitting multiple instances of the data, which exacerbates any existing congestion.

However, by relaxing what is actually required for videoconferencing, and by

adding a more efficient means of data distribution, Internet-based videoconferencing has become a reality. The tools that receive data are designed to compensate gracefully for delay and loss of data, so completely reliable transmission is no longer necessary (although it is still desirable if it is available). Many-to-many communication has been made efficient by the addition of multicasting capabilities to the Internet protocol suite.

Today, a variety of tools exists to exchange multimedia information between many parties in real time over the Internet. In particular, there are several widely available applications, both free and commercial, that enable videoconferencing using the Real-Time Transport Protocol (RTP) [34].

With the advent of these conferencing tools, the need has arisen for ancillary utilities to record and to replay these conferences. Several such tools are being or have been developed.

This paper describes the DPSS Multimedia Recorder/Player (DMRP), which can capture data transmitted via one or more RTP sessions for later playback. It uses the Distributed-Parallel Storage System (DPSS) [20] to store the data. The combination of the DMRP and DPSS makes possible a scalable, robust archival server for such multimedia data.

Chapter II provides background material on both the MBone (its history, protocols, and practices) and the DPSS. Chapter III describes the DMRP's design, including the functionality required (and provided) by the DMRP and

some constraints that influenced that design. Chapter IV details the implementation of the DMRP, including a description of the major classes comprising it and a brief discussion of some non-obvious implementation decisions (the use of threads and the Standard Template Library [36]) and the reasons for them. Chapter V provides a brief performance analysis of the finished software. Chapter VI outlines some of the improvements and modifications that are planned for the DMRP. Finally, Chapter VII presents a summary and concluding remarks.

CHAPTER II

BACKGROUND

Because they are still relatively new, Section II.1 reviews current multicasting practices and protocols, with particular emphasis on the multicast transport of multimedia data. Section II.2 discusses the general purpose, functionality, and architecture of the DPSS.

II.1 Multicasting

Peer-to-peer communication in a computer network can be characterized according to the scope of the intended audience. *Unicasting* denotes the transmission of data from one computer to a single recipient on a network. *Broadcasting* denotes the transmission of data from one computer to all computers on a network. *Multicasting* denotes the transmission of data from one computer to a subset of all the computers on a network.

Multicasting fills a critical gap between unicasting and broadcasting. It makes more effective use of network bandwidth than unicasting by eliminating the need to duplicate data for each recipient; in fact, multicasting permits data distribution in parallel to a much larger number of recipients than is physically possible via

unicasting. At the same time, multicasting permits more discrimination than broadcasting, in principle allowing end nodes to avoid processing data not intended for them.

II.1.1 Multicasting on the Internet

The Internet Protocol (IP) [31] did not originally provide support for multicast. However, in 1989 Steve Deering, then of Stanford University, proposed a set of modifications to IP to allow multicast [6]. The changes were designed to allow implementations that included support for *IP multicast* to interoperate with implementations lacking the modifications.

Under IP multicast, class D IP addresses comprising the address range 224.0.0.0 - 239.255.255.255 are reserved for identifying *host groups*, "a set of zero or more hosts identified by a single IP destination address" ([6], 1). Hosts send data to a host group by setting the destination address of their IP datagrams to the appropriate class D address, and similarly receive data by accepting datagrams sent to that address. Forwarding of IP multicast datagrams is handled by *multicast routers*.¹ Receiving hosts use the *Internet Group Management Protocol* (IGMP) [8] to indicate to the nearest multicast router their

¹. References to "routers" in the following discussion should be understood to mean "multicast routers" unless stated otherwise.

desire to join host groups and so to receive the traffic destined for those groups.² (IP multicast is designed so that a host need not join a host group to send data to that group.)

Broadly, IGMP requires a would-be receiver to signify its desire to receive traffic destined for a particular host group to the nearest multicast router. Once a router has been advised that any host on a connected local network wishes to receive a host group's traffic, the router forwards that traffic to the local network. The router periodically sends a query to all hosts on the local network, requesting that the hosts report which host groups they are (still) interested in receiving. As long as any host reports its interest in a host group, the router continues to forward the group's traffic to the local network. If no host expresses such an interest, the traffic is not so forwarded.

IP multicast is scalable and dynamic because it places the responsibility for group membership on the receiver and the multicast routers; no host—including the sender—need keep information as to the location, or even the existence, of other participants. Multicast routers need not know which hosts on a connected local network are participating in a host group, but only that at least one host is. Furthermore, no matter how many hosts on a local network are members of a

². Multicast routers do not use IGMP to direct multicast traffic between one another: they use one of several multicast routing protocols which will not be discussed in this document.

host group, only one copy of each datagram belonging to the group will be forwarded to the network by the router. Finally, a sender need transmit only one instance of its data, no matter how many receivers there are. Final delivery of data is via broadcast on a LAN, e.g., Ethernet, or multicast if supported by the end host's subnet (e.g., on an ATM network).

The collection of IP multicast-enabled hosts connected by multicast routers and virtual point-to-point *tunnels* over non-multicast-enabled portions of the Internet is called the IP Multicast Backbone, or *MBone* [7]. Eventually, as support for IP multicast becomes ubiquitous, the MBone will cease to be a distinct subset of the Internet.

Note that IP multicast only provides the barest essentials for communication; like unicast IP, IP multicast requires higher-level protocols to be useful to applications. These protocols must address not only problems analogous to those addressed by non-multicast protocols, such as reliable end-to-end delivery, but also issues pertinent to multicast in general, such as the monitoring of group membership, and issues relevant to IP multicast in particular, such as the apportioning of scarce resources (i.e., IP multicast addresses). RTP, discussed in Section II.1.2, provides multicast data transmission services to applications that can survive some unreliability, including misordering of data. Other protocols, discussed in Section II.1.3, address a different subset of the higher-level protocol issues: they attempt to impose some order and meaning to

host groups by associating information with them and disseminating the information as needed. These protocols are critical to the deployment of *session directories*, which are tools that provide meaningful descriptions of how host groups are currently being used—e.g., "224.2.153.144, port 43654 is being used for H.261 video data exchange by the NASA Space Shuttle program"—and which provide enough data to enable a user to join the host group using an appropriate tool for the data being exchanged. Yet another set of protocols, also discussed in Section II.1.3, are intended to govern the actual performance of multimedia data transmission, allowing control of data rates and dynamic, high-level coordination of sets of multimedia data streams.

II.1.2 A New Protocol: RTP/RTCP

II.1.2.1 Limitations of UDP and TCP

By itself, IP multicast is insufficient to support end-to-end data transport. As with any IP datagrams, IP multicast datagrams may be lost or may arrive out of order [6]. Thus for IP multicast to be useful, some higher-level protocol must supply the necessary error detection—and if necessary, error compensation—facilities. In addition, the higher-level protocol should support multiplexing of distinct multicast sessions between two or more hosts, and possibly support some organization of the data (e.g., by partitioning it into frames or segments).

UDP and TCP, the most widely used transport-level protocols operating over

IP, were designed to facilitate, respectively, message- and stream-oriented data transfers over a network or internetwork. UDP offers applications multiplexing (i.e., the use of multiple, distinguishable communication endpoints ("ports") on a single host) and some data integrity checking (error detection), while TCP provides reliable, ordered delivery (including error correction) in addition to multiplexing and integrity checking. Both protocols impose some organization on the data they transport: UDP packages data in datagrams, while TCP treats data as ordered streams of bytes.

TCP, however, is designed for unicast communication. It establishes a duplex connection between two and only two hosts: its connection establishment, error detection and error correction algorithms are designed around this duplex communication. TCP would have to be drastically modified to support simultaneous communication between more than two hosts.

Lacking TCP's extensive error correction mechanisms, UDP might seem an appropriate candidate for an IP multicast transport protocol. However, UDP does not itself provide enough information to determine whether datagrams are missing.

If IP multicast is to be used to deliver real-time data like video, the higher-level protocol must provide the necessary mechanisms to ensure that the receiver can present the data as the sender intended in time (i.e., it must provide synchronization mechanisms). It is important, for instance, to present video to an

end user at the correct frame rate, just as it is important to preserve the pauses in speech since they may signify as much as the words. Neither TCP nor UDP preserves this data-based timing.

Human beings can tolerate some degree of loss in an audio or video stream: moviegoers, for example, endure splices in film reels without significant loss of understanding. Complete reliability, therefore, is not required, only the ability to detect loss. However, TCP does not allow reliability to be sacrificed, while UDP has no way of detecting loss.

The *Real-time Transport Protocol* (RTP) [34] is a stream-oriented, application-level datagram protocol. Together with the *RTP Control Protocol* (RTCP), RTP provides enough information for a receiver to reconstruct the stream, or to detect gaps within the stream.³ RTP and RTCP also provide enough information to calculate the rate at which the datagrams were sent. However, they deliberately do not define a mechanism to compensate for lost data: the designers believed that application developers might wish to use various strategies to recover from errors.

Although RTP and RTCP were designed to support "multi-participant multimedia conferences" ([34], 3), the protocol is not explicitly limited to such

³. "RTP" is often used to denote both the data transport protocol and the combination of RTP and RTCP, with the meaning being clear from context. In this paper, where ambiguity may exist, both protocols are explicitly mentioned.

applications. However, when discussing RTP/RTCP usage below, examples will be in the context of such conferences.

It should be noted that RTP/RTCP provides neither traditional port-based multiplexing (as used by UDP and TCP) nor data integrity checking, relying on an underlying protocol for these services. The protocol's designers envisioned that UDP would be used for this purpose, although any equivalent protocol could be used. RTP does require multiplexing of a different sort, however: each data source in a host group must distinguish itself using a unique identifier, the SSRC ID (see Section II.1.2.2).

RTP and RTCP were designed to provide a minimal common basis for real-time communication between a variety of applications while allowing those applications the flexibility to choose how, or whether, to enhance the basic set of offered services.

In this paper, the protocol specification for RTP/RTCP is sometimes referred to as "RFC 1889," the identifier assigned to it by the standards body that issued the specification.

II.1.2.2 RTP

RTP facilitates the dissemination of real-time data in a multicast group by providing key services that IP multicast lacks. It identifies and distinguishes session members, and timestamps data so receivers can reconstruct a sender's

data stream in time as well as space.

An *RTP session*, in terms of IP multicast,⁴ embodies all participants communicating via a particular multicast address and pair of port numbers (one for RTP, the other for RTCP). In a multimedia conference, each medium typically warrants its own session; thus audio and video data are carried in different RTP sessions. (Discussion is under way as to how mixed-media encodings—those that carry both audio and video, for example—should be treated by the protocol.)

Central to RTP is the concept of the synchronization source. A *synchronization source* (SSRC) is the origin of a stream of RTP data packets and defines both the ordering and the timing of the data stream it transmits. Each audio or video source in an RTP session, for example, would be a separate SSRC. Since in a typical multicast group any participant may be a data source, every participant usually is considered an SSRC. Each SSRC must uniquely identify itself in a session by choosing a random *synchronization source ID*. The SSRC ID allows receivers to associate packets, arriving in random order, with the streams of which they are a part. Note that using packets' source addresses to multiplex between sources is inadequate, since a single host may originate multiple streams.

An SSRC need not be a source of raw data, such as the input from a

⁴. The term has a slightly different meaning if RTP and RTCP are used for unicast.

microphone or camera: it may also be an application that filters an existing RTP data stream or combines several such data streams in some way. The input to such an application, known as a *mixer*, might require timing adjustments before the data could be retransmitted. The mixer would then be the (new) data stream's synchronization source, because the timing and sequence of the data would no longer reflect that of the original data sources, or *contributing sources* (CSRCs). A mixer identifies the CSRCs whose packets it is filtering so that a receiver may correctly attribute the packets' originators.

RTP segments a raw data stream into packets and prepends a 12-byte fixed header of protocol information (see Figure 1) followed by zero or more

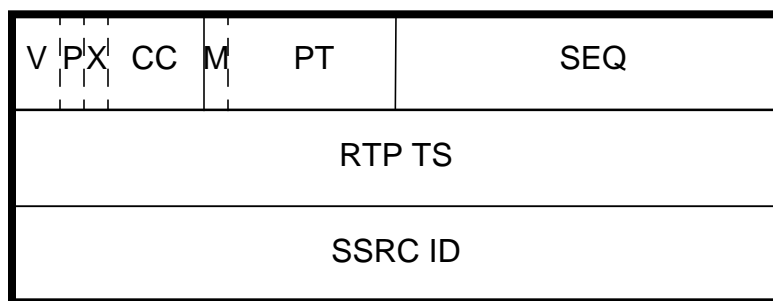


Figure 1. RTP Fixed Packet Header

contributing source identifiers and possibly one header extension. (Header

extensions are beyond the scope of this discussion.)

The *V* field (version; 2 bits) indicates the protocol version number; the current version is 2.⁵ The *P* field (padding; 1 bit) indicates whether the packet contains padding following the payload, while the *X* (extension) bit, if set, indicates that the fixed header is followed by one header extension. (Padding, like header extensions, is not relevant to this brief discussion of the protocol: the DMRP treats both opaquely.) The *CC* field (CSRC count, 4 bits) is the number of 32-bit contributing source identifiers that follow the fixed header; zero is a valid count.

The *M* field (marker; usually one bit) is interpreted according to the *profile*⁶ under which the application is operating (see Section II.1.2.4); generally, it marks "significant events such as frame boundaries . . . in the packet stream" ([34], 11). The *PT* field (payload type; usually 7 bits) tells an application what kind of data are in the packet; the interpretation of the RTP timestamp varies according to the payload type. The size and permitted values of each field may vary depending on the profile, but together they are constrained to occupy only one octet.

The *SEQ* field (sequence number; 16 bits) identifies the order in which the packets were transmitted. It increments by one for each RTP data packet sent.

⁵. RTP version 0 (RTPv0) was used by one of the earliest MBone applications, vat. RTPv1 was an experimental version that was not widely deployed.

⁶. A profile is one set of RTP parameters typically agreed upon for transport of a particular medium or media, e.g., CD quality audio.

RTP does not ensure in-order presentation of packets, so the sequence number allows an application to detect an out-of-order packet. A transmitter's initial SEQ value is randomly chosen to enhance encryption security, although the protocol does not require the use of encryption.

Because the SEQ field only comprises 16 bits, it can only take on 65,536 unique values. During a long-lived RTP session, it is likely that more than this number of packets will be transmitted by a continually sending source, so the source must reuse sequence numbers. The protocol specification requires that session members note each time the SEQ has wrapped around, i.e., used all 65,536 values; each such sequence number wrap is called a *cycle*.

The *RTP TS* field (RTP timestamp; 32 bits) "reflects the sampling instant of the first octet in the RTP data packet" ([34], 11). The RTP timestamp does not represent the actual or wallclock time: rather, it represents the passage of time on a logical clock. How this timestamp corresponds to wallclock time is format-dependent (see Section II.1.2.4). Receivers use the timestamps to synchronize presentation of data to the user, reproducing the playout rate of the source. Like the SEQ, the initial value of the RTP TS is randomly chosen.

The *SSRC ID* field (32 bits) is the SSRC's randomly chosen identifier for the duration of the RTP session. Each SSRC ID must be unique during a session: if two SSRCs choose the same SSRC ID, one or both must select a new ID. An

SSRC must also change its SSRC ID if it changes its source transport address.

If the packet was generated by a mixer, the CSRC IDs follow the SSRC ID field.

The structure of the RTP fixed packet header does not vary across profiles (see Section II.1.2.4) or for different payload types governed by a profile.

II.1.2.3 RTCP

According to RFC 1889, RTCP performs three main functions:

1. It provides "feedback on the quality of the data distribution." This feedback can be used by other participants (e.g., a sender might choose to reduce its bandwidth because of poor reception by other members of the session) or by third-party monitors that may act on that information (e.g., a network provider might elect to change bandwidth constraints for a session).
2. It carries an RTP source's *canonical name*, which is a persistent identifier for the source, unlike the SSRC ID which may change during the course of a session. The canonical name allows receivers to associate a meaningful identifier with a source, especially between concurrent sessions. This cross-session persistence permits association between different media, with the foremost example being synchronized audio and video.

3. Because every session participant must send RTCP packets, participants can determine the total number of session members dynamically. The number of session members is one parameter determining the RTCP packet transmission rate, which must be controlled to allow RTP/RTCP to scale to a large number of participants. Hence RTCP contributes to adjusting its own performance.⁷

The structure of the first four bytes (32 bits) present in all RTCP packet headers is shown in Figure 2.



Figure 2. RTCP Common Packet Header

The *V* and *P* fields are the same size and have the same meaning as the fields of the same name in the RTP data packet header, and *V* currently also has the same value, 2.

The *C* field (count; 5 bits) is a count of subsequent items in the packet. Different RTCP packet types carry blocks of per-source data; *C* indicates how many such blocks are part of the packet, i.e., how many sources are being

⁷. RTCP can also "convey minimal session control information." However, it is more common to convey this information via the protocols discussed in Section II.1.3.

reported on in this packet. (For one type of RTCP packet—the APP packet—the C field is called the *subtype* field and may be used in an application-specific way rather than as a count. The field is the same size in either case.)

The *PT* field (packet type; 8 bits) denotes the packet's type; there are five types (see Table 1). The *LEN* field (length; 16 bits) gives the RTCP packet's length in 32-bit words minus one, including the header and any padding. Defining LEN in this way provides for slightly greater efficiency during packet validation and avoids a possible infinite-loop error when scanning compound packets; see RFC 1889 for details.

Table 1 shows the five types of RTCP packets and the value of the PT field for each.

Table 1. RTCP Packet Types

Packet Type	Abbrev	PT value
Sender report	SR	200
Receiver report	RR	201
Source description	SDES	202
Goodbye	BYE	203
Application-defined	APP	204

Sender report (SR) packets are sent only if the session member transmitted

data since its last or next to last report (either SR or RR). The structure of an SR packet is shown in Figure 3.

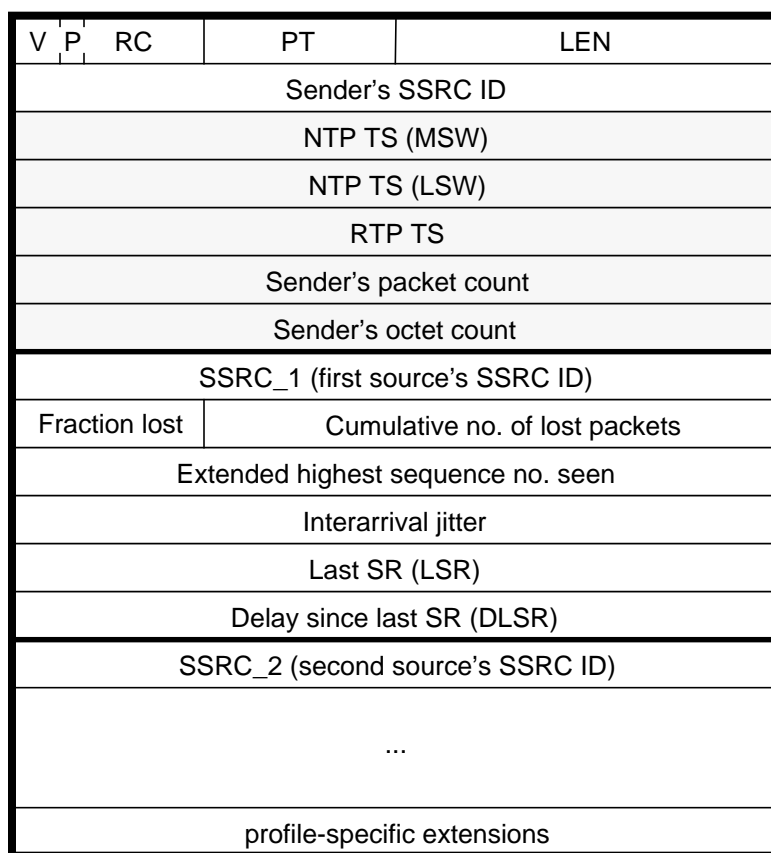


Figure 3. RTCP SR Packet

Following the RTCP common header are the sender's SSRC ID (32 bits) and 20 octets (five 32-bit words) of sender-specific information (shaded in Figure 3), the most important of which for this discussion are the NTP and RTP timestamps. The NTP (Network Time Protocol) [27] timestamp represents the wallclock time at which the SR was sent, while the RTP timestamp is the representation of the NTP timestamp in the same units as the RTP data

timestamp. The two timestamps allow inter-media synchronization for SSRCs whose timekeeping is coordinated by NTP. The NTP timestamp is represented as two 32-bit words, with the most significant word (MSW) preceding the least significant word (LSW) in the header.

Zero or more *reception reports* follow the sender-specific data. Each represents an SSRC from which the sender received data since transmitting its prior SR or RR. A reception report includes, *inter alia*, the fraction of packets lost since the last report, the highest sequence number received, an estimate of the variance in data packet interarrival times (interarrival jitter), and the delay since the last SR from the SSRC. The C field in the RTCP common header is defined as the *reception report count* (RC) for the SR (and RR, below).

Finally, an SR may contain a profile-specific extension, discussion of which is beyond the scope of this overview of the protocol.

Receiver report (RR) packets are identical to SR packets except that they do not contain the 20-octet block of sender-specific information, and their PT field is set to 201.

Source description (SDS) packets convey identifying information for the

named packet source(s). The structure of an SDES packet is shown in Figure 4.

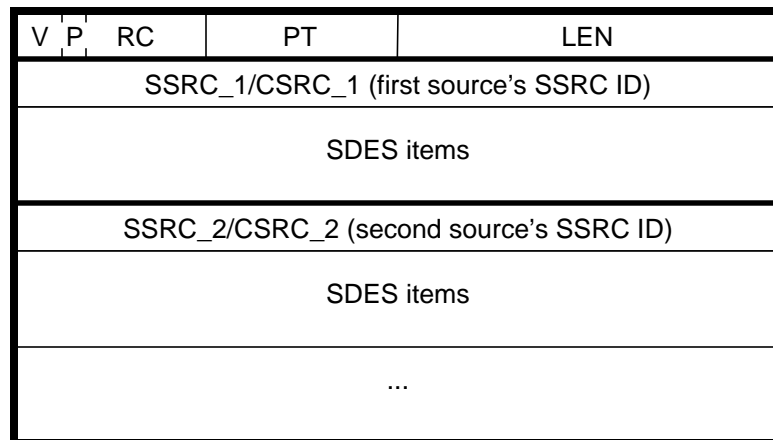


Figure 4. RTCP SDES Packet

SDES packets consist of the RTCP fixed header (with a value of 202 for the PT field) followed by zero or more units known as *chunks* which in turn each consist of an SSRC ID (or a CSRC ID if the reporter is a mixer) and one or more SDES *items*. An SDES item is made up of a type, a length, and data, as shown

in Figure 5.

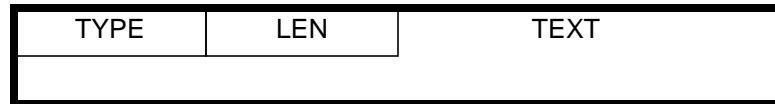


Figure 5. RTCP SDES Item

SDES item types are listed in Table 2.

Table 2. RTCP SDES Item Types

Item Type	Value
CNAME	1
NAME	2
EMAIL	3
PHONE	4
LOC	5
TOOL	6
NOTE	7
PRIV	8

The LEN field (item length; 8 bits) in an SDES item indicates the size of the data in octets, not including the two octets of the item type and LEN fields.

Within a chunk, items are "stacked" contiguously and are not required to end on 32-bit boundaries. However, each chunk as a whole must begin and end on a 32-bit boundary so there may be one or more null octets after the last item in a

chunk. The DMRP, like most existing MBone applications, encodes SDES item data in US-ASCII format.

From the standpoint of a quick survey of the protocol, the most important SDES item type is the CNAME. SDES CNAME chunks carry the source's *canonical end-point identifier*, which is a unique identifier for a source within a session that, unlike an SSRC ID, is guaranteed to remain constant for the session's duration. One of its intended uses is as a means of cross-media synchronization, which implies that the CNAME should be fixed across multiple related RTP sessions. The protocol specification strongly recommends that the CNAME be derived algorithmically (i.e., automatically) and should take the form "user@host". At minimum, an SDES packet must carry the sender's CNAME, and the CNAME is the only SDES item type that all RTP/RTCP applications must support.

The TOOL item allows an application to identify itself to other session participants. Current tools often include version identification information as well.

The RTCP common header's C field is defined as the *source count* (SC) for an SDES packet and denotes the number of chunks in the packet.

Goodbye (BYE) packets indicate that a source is no longer active, i.e., that it has left the RTP session. *Application-defined* (APP) packets are for experimental use: they obviate the need to register new RTCP packet types while the new types' utility is being tested. Neither of these is important to

understanding RTP/RTCP in the context of the DMRP, so they will not be discussed further.

RTCP is designed to scale to large sessions without overwhelming either the associated RTP data traffic or the available network bandwidth. To reduce the protocol's overhead, several RTCP packets are combined into a single *compound packet* before being sent. More importantly, the frequency of a source's RTCP packet transmission is limited so that the aggregate rate of all members' RTCP traffic consumes no more than a small fraction of the bandwidth available to a given RTP session, with the suggested fraction being 5%. Whatever the value, however, it must be fixed for a given profile (see Section II.1.2.4), as each participant independently calculates its RTCP inter-packet transmission interval based on the available RTCP bandwidth.

The share of the bandwidth allotted to a given participant, together with the average amount of data in an RTCP packet, determine the average interval between that participant's RTCP packets. As more sites join an RTP session each site must send RTCP packets less often, assuming that the session's total available bandwidth does not change.

II.1.2.4 Protocol Customization

RFC 1889 does not completely specify RTP and RTCP in the same way as their respective RFCs specify, e.g., TCP or IP. The protocols' designers

envisioned that RTP/RTCP would be implemented within applications rather than as separate modules or libraries, using the requirements of RFC 1889 as a basis for minimal interoperability.

A complete specification of the protocols must include a *profile* and *payload format specifications*. A profile characterizes an entire class of applications; among other things, a profile corresponds payload type identifiers to payload formats and, for each payload type, defines the RTP timestamp clock rate to be used. Most existing RTP-based tools operate under the profile for audio and video applications, RFC 1890 [33]. A payload format specification describes how a particular kind of data format is to be carried by RTP. There are payload specification documents for various audio and video formats such as H.261 [40], JPEG video [1], MPEG1/MPEG2 video [16], and redundant audio [29].

II.1.3 Other Internet Conferencing Protocols

RTP and IP multicast are components of what is an emerging set of protocols to provide multimedia conferencing over the Internet; collectively, these protocols have been dubbed "the Internet Multimedia Conferencing Architecture" by the Internet Engineering Task Force (IETF) [13], which has been developing them. The conferencing architecture's protocols fall into two categories, conference management and data distribution. RTP is an example of an unreliable distribution protocol; the IETF's model also encompasses reliable (multicast)

protocols such as SRM (see Section II.1.4).

Conference management protocols are concerned with the setup and high-level control of presentations. For this discussion, a *presentation* or *conference* is a "set of one or more streams presented to the client as a complete media feed," where a *stream* is a "single media instance, e.g., an audio stream or a video stream as well as a single whiteboard or shared application group" [35]; an RTP session as defined in Section II.1.2.2 is thus an instance of a stream. In the IETF architecture, conference management decomposes into conference setup and discovery, and conference course control (i.e., managing the bandwidth and possibly the membership of a presentation). While RTCP can be used for limited conference course control—this is a fourth, optional function according to RFC 1889—more sophisticated protocols are being developed specifically for this purpose.

For a limited number of participants, conferences can probably be arranged by email or telephone. However, such informal methods clearly do not scale to a large number of participants, nor to conferences whose participants cannot be known in advance. *Conference setup and discovery* protocols address the problem of creating, describing, and finding presentations. These protocols fall into two domains. The description of a presentation—which can be considered the aggregate of the descriptions of the constituent media—is logically distinct from the mechanism by which that description is made available to potential

receivers.

The *Session Description Protocol* (SDP) [14] encodes all the relevant information about a presentation, where "all the relevant information" means that SDP provides enough information for a receiver to participate in the presentation.⁸ SDP includes both presentation-wide parameters, e.g., the presentation's name, and media-specific parameters such as the media type (audio, video, etc.) and address/port(s). For maximum flexibility in handling, SDP data are entirely textual, with US-ASCII being a subset of the accepted encoding scheme.

SDP describes a presentation, but does not attend to the other half of conference setup and discovery, the distribution of the information. The IETF, in an overview of its proposed conferencing architecture [13], notes four protocols by which SDP data may be disseminated. Two of these, the Hypertext Transfer Protocol (HTTP) [9] and the Simple Mail Transfer Protocol (SMTP) [32] are well known and in wide use on the Internet for exchanging much more than conference information, so they will not be discussed here. The *Session*

⁸. The IETF protocol documents use "session" inconsistently. The conference control protocols such as SDP, SAP and SIP use "session" to denote what was defined as a "presentation" above, while RTP/RTCP defines a "session" in terms of an address and port number pair associated with a single medium. In this discussion, "session" will always have the RTP/RTCP meaning, and "presentation" will be used to describe a set of multiple sessions. Following this usage, SDP, for example, would more properly be called the "Presentation Description Protocol."

Announcement Protocol (SAP) [12] and *Session Initiation Protocol* (SIP) [15] are specifically designed to carry presentation information. SAP and SIP represent two different distribution styles: "announcement" and "invitation."

SAP multicasts SDP-encoded packets to well-known multicast addresses and ports. SAP permits an organization to restrict SAP packets to arbitrarily defined administrative domains by choosing particular local addresses, a practice known as *administrative scoping*, or to use a reserved address and port for more generally accessible *time-to-live scoped* announcements (see below). The use of well-known addresses and ports (whether administratively scoped or not) permits the implementation of distributed directory tools—the session directories mentioned in Section II.1.1—that list presentations announced via SAP; the best-known such tool is sdr (see Section II.1.4). SAP allows anyone to discover a presentation and, via SDP, provides enough information to join it.

A *time-to-live* (TTL) mechanism determines the range of the packets' distribution. All IP packets contain a TTL field which is decremented (by one) by each router through which the packet passes; a packet with a TTL of zero is destroyed, preventing it from circulating indefinitely. In addition, a multicast router compares each packet's TTL to a *threshold* value. Each multicast router interface and tunnel is configured with such a threshold, which denotes the minimum TTL that a packet must possess to be forwarded through the interface or tunnel. Proper use of TTLs is necessary to ensure that SAP, RTP/RTCP, and

other multicast protocols can scale to large numbers of presentations, as the traffic of non-global presentations (e.g., a high-bandwidth video conference within an organization) must not be allowed to congest network links outside the presentations' intended scope. TTL-scoped SAP packets are assigned the same time-to-live value as the presentations they advertise.

SAP is analogous to an advertisement that invites the public at large to attend a meeting (although SAP announcements can be encrypted so that only specific parties can decode them and participate in the announced conference). SIP, in contrast, provides a means to invite select parties to a presentation in a manner reminiscent of a telephone conference call. SIP is a complex protocol that supports "some or all of four facets of establishing multimedia communications" [15]: (1) locating the invitee, (2) negotiating what media and media parameters are possible and appropriate for the invitee, (3) determining whether or not the invitee wants to join the presentation, and (4) establishing the "call parameters" of both the inviter and invitee, as a prelude to the invitee's actually joining the conference. Like SAP, SIP typically will use SDP to encode the presentation information.

What the IETF terms "conference course control" [13] at present refers to *quality of service* (QoS) guarantees and dynamic control of active presentations. QoS guarantees (e.g., loss rate) are handled by the *Resource ReSerVation Protocol* (RSVP) [2]. RSVP is used "to request specific qualities of service from

the network for particular data streams or flows," as well as "to deliver [QoS] requests to all nodes along the path(s) of the flows and to establish and maintain state to provide the requested service" ([2], 4). RSVP is intended to be used in both routers and end hosts. Like IGMP (Section II.1.1), RSVP is receiver-oriented: receivers must issue RSVP requests, rather than senders. This permits RSVP both to scale to potentially large numbers of receivers, and to adapt to group membership changes.

RSVP requests propagate along the same path that data will take from the sender to the receiver, but in the opposite direction. At each intermediate node, RSVP causes sufficient resources to be allocated for the QoS and bandwidth requested if possible. If insufficient resources are available, or if the requester is not authorized to obtain the resources, an error is returned to the requester. By default, a failure to obtain resources at a node does not release resources successfully allocated at other nodes closer to the receiver, however, because the receiver may be willing to settle for having the desired QoS along as much of the path as possible.

All of the conference management protocols described so far operate prior to the start of the presentation; even RSVP, which affects the presentation's delivery to receivers, performs the bulk of its work before any data are transmitted. The *Real Time Streaming Protocol* (RTSP) [35] functions as a control protocol for continuous media streams (e.g., audio and video) while the

presentation is active. RTSP has three functions: (1) it can initiate retrieval of streams from a media server, (2) it can "invite" a media server to join an existing presentation, and (3) it can notify a receiver of the existence of newly available media (e.g., a media server can notify receivers that a new video source is available, perhaps corresponding to a new conference participant) for an existing, usually live, presentation. A *media server* is defined as a "network entity providing playback or recording services for one or more media streams" ([35], 8). Thus the first two functions explicitly apply to media recording and playback devices, making RTSP "a 'network remote control' [protocol] for multimedia servers" ([35], 5). RTSP defines traditional VCR-like commands such as "PLAY," "PAUSE," and "RECORD" that control the allocation and usage of resources on a server.

It should be noted that none of the conference management protocols described, including RTP/RTCP, requires the use of any of the other protocols, although they are designed to interoperate when available. The DMRP does not use any of the protocols except RTP/RTCP, although it does indirectly take advantage of SDP-encoded presentation descriptions made available via sdr; see the discussion of the DMRP's configuration file in Section IV.3.

Finally, the IETF also acknowledges [13] the need to control the allocation of IP multicast addresses and ports to avoid collisions between presentations. However, no consensus has been reached on the proper means of doing so.

One suggestion [15] is for address allocation to be managed by SAP, but others argue that where appropriate, "[the] media server [should pick] the multicast address and port" [35]. In practice, most if not all advertised presentations appear to have their address(es) and ports chosen by sdr, described in Section II.1.4, via a mechanism that is not yet documented in any IETF specification.

II.1.4 Existing MBone applications

A number of RTP-based conferencing tools are in use on the MBone to communicate audio, video, ASCII text, and PostScript (R), among other data formats. In addition, there is a VCR-like tool for recording and playback of conferences as well as a session directory tool.

The best-known applications for audio conferencing are the *Visual Audio Tool* (vat) [28] and the *Robust-Audio Tool* (rat) [23]. Both vat and rat establish or join an RTP session for audio traffic. A user can monitor session membership, adjust the volume of sent and received audio, and control the rate at which his own audio is sent, all via graphical interfaces.

vat's existence actually predates RTP: in fact, vat's original communication protocol was the basis for the first version of RTP, RTPv0. vat has evolved to become a useful multicast network debugging tool as well as a conferencing tool since a user can monitor several important RTP traffic statistics in real time. rat, perhaps in the interest of minimizing confusion for the novice user, eschews most

of vat's diagnostic options.

vat's video complement is the *Video Conferencing Tool* (vic) [26]. vic establishes or joins an RTP session for video traffic. As with vat, a user can extensively control and adjust vic's behavior, including the data transmission rate and image quality. Also like vat, vic allows the user to monitor RTP traffic statistics and session membership in real time.

The *whiteboard tool* (wb) [25, 10] allows conferees to share and to modify whiteboard-like images; wb can thus serve as a distributed notepad during a conference or as a viewgraph viewer during a lecture. The images can be divided into *pages*, each of which can be created or drawn upon by any session member. A member's wb drawings generate messages called *drawops*. Each drawop is timestamped and assigned a sequence number; together with the member's identifier, these ensure that every drawing is uniquely named and ordered within the context of that member's data stream. The drawops are multicast to the entire session.

wb differs from other MBone tools because it requires a reliable multicast communication protocol: it cannot tolerate data loss. wb implements such a protocol, dubbed *Scalable Reliable Multicast* (SRM) [10]. SRM's error recovery is receiver-driven, requiring receivers explicitly to request retransmission of missing data. However, any member of the session—e.g., one that is "close" to the requester—may respond to such a request, not just the original sender. SRM

includes mechanisms to reduce the likelihood of redundant requests and responses that accompanies such a flexible recovery policy. Because these mechanisms introduce some delay, SRM is probably not well suited to all, or perhaps any, contexts in which RTP is currently used. (It should be noted that SRM is a separate protocol from RTP.)

The *MBone VCR* [17] serves the same purposes for MBone sessions as a conventional VCR serves for broadcast television. It can record one or more RTP sessions, saving them to files on the local filesystem, and can later replay the data as MBone sessions. It can be given a configuration file describing the sessions and a separate command file controlling when it starts recording or replaying, thus allowing automated use. (The DMRP accepts the same configuration file format; see Section IV.3.)

sdr is a session directory of the type described in Section II.1.1. It disseminates announcements of MBone conferences, allowing prospective participants to learn about conferences that may interest them. Conference organizers register their desire to transmit one or more RTP sessions, specifying how many media sessions are needed, the payload format for each, and other relevant information, including a human-readable description. *sdr* encodes the presentation information via SDP and promulgates it using SAP; it receives, decodes, and displays SDP messages from remote *sdr* instances for the user; and it can launch the appropriate MBone conferencing tools for a multimedia

session if they can be found.

The foregoing is meant to illustrate the types of tools that are in common use on the MBone. Other applications exist or are under development, but an exhaustive survey of MBone-related software is beyond the scope of this paper.

II.2 The Distributed-Parallel Storage System

II.2.1 Purpose

The *Distributed-Parallel Storage System* (DPSS) was created to serve applications requiring high-speed access to very large data sets via high-bandwidth data streams. The assumptions underlying the DPSS' existence are first, that the data of interest cannot readily be accommodated, even temporarily, within disk space and memory locally available to the client; and second, that the client requires high-speed, perhaps near-real-time, access to the data. Moreover, because the connotation of "very large" is subject to change over time, the DPSS is intended to scale to accommodate nearly arbitrarily large data sets.

The prototype DPSS client application, *TerraVision* [24], uses data sets that are on the order of 1-10 GB in size. The data represent terrain imagery which TerraVision renders and displays in response to user actions. The user thus appears to be navigating through the terrain in three dimensions in real time. To create such a realistic dynamic visualization, TerraVision requires 300-400 Mb/s

of data.⁹

In principle, the DPSS can be modified to support multiple lower-bandwidth data streams whose aggregate bandwidth is on the order of a single TerraVision data stream. Thus, a DPSS potentially could support dozens or hundreds of MBone media streams.

The DPSS is discussed in more detail in [20], [21], and [39].

II.2.2 Functionality

At heart, the DPSS is a network-based block server cache. The unit of data exchange between clients and the DPSS is a block; any other granularity required (e.g., obtaining a particular byte within a block, or aggregating multiple blocks as a single data element) is the responsibility of the client application.

Blocks are organized at two levels. A *data set* is a logically associated collection of blocks representing a distinct logical name space. The DPSS assigns to each data set a unique *set ID*. Within a data set, each block has a unique *logical block name*. An individual block on the DPSS is thus completely identified by its set ID and logical block name.

The DPSS achieves high-bandwidth data streaming by exploiting the parallelism available via multiple disk servers, each operating multiple disks and

⁹. TerraVision cannot absorb data at that rate due to limitations in the rendering hardware on the SGI Onyx computer on which it has been run. To date, TerraVision has been able to absorb a maximum of 80 Mb/s.

attached to a high-bandwidth network. A client's requests for data are broken down into requests for individual blocks which were distributed among the DPSS hosts during loading (see below for more details on data loading). Multiple block requests can thus be satisfied in parallel by multiple disks on multiple hosts seeking and reading simultaneously. As soon as a block is available, it is delivered to the client. On a high-bandwidth network, multiple blocks can be sent simultaneously. Although blocks may—indeed, likely will—arrive out of order, each is "tagged" with its set ID and logical block name, allowing the client to determine what to do with the block.

To date, DPSS clients have been designed to request multiple blocks at a time, each of which can to some extent be operated on independently, and this has influenced the evolution of the DPSS. Most of the target data have been imagery, with the most prominent being the terrain visualization data used by TerraVision and the cardioangiography "films" used by the prototype medical imaging application *vp/layer* [38]. However, there is no inherent design or implementation bias in favor of image data: the DPSS is intended to accommodate any large data collections, and more recently it has been proposed as a component for caching high-energy physics data [11] and as a general-purpose staging cache for a high-performance storage system. Each application has its own strategy for coping with out-of-order data. In presenting the data to a user, TerraVision "fills in" the screen as each block arrives, while

vplayer buffers blocks until a complete frame of the film is available. Neither application relies on in-order delivery, nor could either application present its data usefully if forced to wait for blocks to be delivered sequentially.

II.2.3 Architecture and Client Use

A DPSS is composed of multiple low-cost, medium-speed Unix workstations, each of which typically has multiple disks and at least one high-speed network interface attached.

Each workstation runs two types of DPSS software. The *disk server* is a Unix daemon that manages both reading from all the disks on a host as well as writing data to the network (i.e., to a client). It also establishes and maintains a cache in memory of recently-requested blocks. The *scribe* is a Unix daemon that writes data to a single disk on a single host. A single workstation may have multiple scribes running, but only one disk server.

One host in a DPSS is designated the *master*. The master loosely coordinates the activities of the disk servers: it presents lists of requested blocks for the servers to process and it maintains much of the DPSS' global state. The master is also the initial point of contact for clients and manages all control communication between clients and the DPSS. Clients establish an initial connection to the master, then typically establish connections to as many scribes

or disk servers as necessary.

When reading data from the DPSS, clients send lists of logical block names, representing the desired data blocks, to the master. The master validates the list as necessary, verifying the existence of the requested blocks.¹⁰ Each valid block name is translated to a request for some number of bytes from a certain physical location specified by server host, disk, and disk offset, then forwarded to the appropriate disk servers for satisfaction. The disk servers retrieve the requested data, either from disk or from their memory cache, and write the data to the client.

Blocks read from disk are added to a memory cache maintained by the server. Caching assists clients having high request rates, e.g., TerraVision, because by default each new set of block requests causes the DPSS server to disregard, or flush, any unsatisfied requests from the prior set of requests. Since a block may have been read from disk but its still-pending request flushed before it could be sent to the client, caching the block keeps it available for subsequent access should the client re-request it, as is likely with TerraVision.

When writing data to the DPSS, clients send data blocks directly to the scribes, sending only metainformation—specifically, the correspondence

¹⁰The master may also enforce security policies, authenticating client requests and verifying access authorizations.

between each block's logical name and its physical location—to the master.

In each case, data are transferred directly between the client and the relevant disk server or scribe, preventing the master from becoming a bottleneck in the data transfer. However, the master monitors all data transfers by virtue of its role as the DPSS name-translation module, corresponding logical block names and physical locations. (By providing this mapping service, the DPSS allows clients to refer to their data in more meaningful ways, while simultaneously isolating clients from the details of physical location. Such isolation, together with a redundant storage strategy, allows the master to perform transparent fault recovery at runtime by redirecting requests away from a failed disk or host to backup hosts and/or disks.)

In order to ensure maximum efficient use of the DPSS—i.e., to achieve the greatest possible parallelism of operation at the server and disk levels when reading, and to achieve the highest possible network bandwidth from the disk servers to the client—client data can be written to the DPSS with special regard for characteristic or expected data access patterns. Much effort, for example, has gone into optimizing block placement for TerraVision data sets based on expected access patterns [5]. However, in practice it appears that random placement with a high degree of parallelism works nearly as well in most cases.

The data rates of current MBone sessions are several orders of magnitude smaller than typical TerraVision data rates. However, the MBone is not

intrinsically limited to these low average data rates: rather they are an attempt to accommodate the many portions of the Internet connected via low-bandwidth links such as modems. MBone conferences can and will consume more bandwidth as it becomes available. Indeed, MBone sessions of multi-megabit data streams have been and are being held within high-speed, high-bandwidth networking environments, e.g., BAGNet [41].

CHAPTER III

FUNCTIONALITY AND DESIGN ISSUES

Conceptually, the DMRP can be considered an RTP packet filter and generator, coupled with a mechanism for storing these packets on, and retrieving them from, the DPSS. The two functions are largely separable, linked only by the data being stored or played back, and the DMRP's design reflects this natural separation.

III.1 RTP/RTCP Packet Management

The DMRP is intended to be a fully interoperable RTPv2 application, which means that it must implement the protocol requirements of RFC 1889 and RFC 1890. Reviewing these documents, it became clear that RTP and RTCP packets would require separate filtering mechanisms since the packets have quite different internal structures. However, corresponding RTCP and RTP streams must share a significant amount of state information. These two imperatives determined the DMRP packet manager's basic conceptual design, as illustrated

in Figure 6:

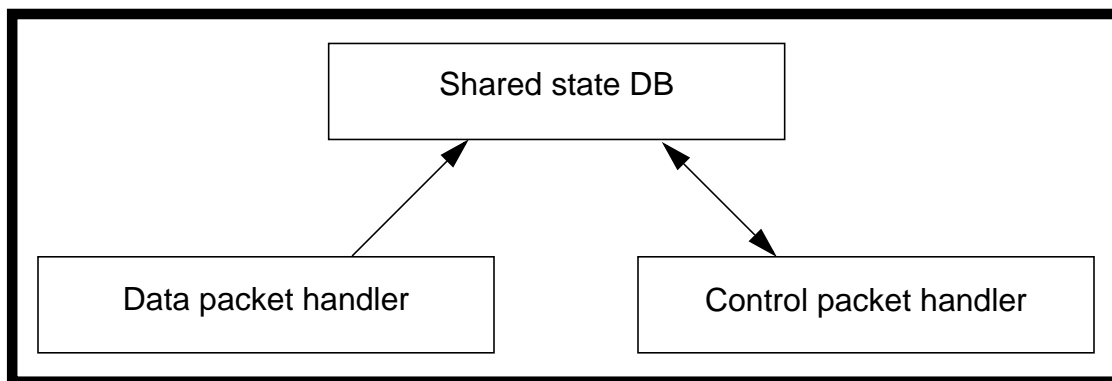


Figure 6. DMRP Packet Manager (conceptual design)

The data and control handlers process RTP and RTCP packets, respectively, as their names suggest. The shared state database is the mutually accessible repository representing the data to which both handlers require access. Note that the shared state includes the information conveyed by every other participant's RTP and RTCP packets, so the database is organized by source.

The arrows in Figure 6 denote the flow of data. The data handler only updates the shared state; the control handler both updates the shared state and accesses it to generate the application's RTCP packets.

III.2 DPSS Constraints

In truth, the DPSS is a somewhat awkward match as a storage device for current MBone data because the average rates of data produced and consumed

by current MBone tools are so small. The DPSS was designed for optimal behavior in an environment where significantly greater average bandwidth usage is the norm: as a result, DPSS blocks are on the order of 64 KB. RTP packets vary in size according to the type of data they carry, but the largest packets at present are only on the order of 1-2 KB.

The disjunction between the granularity of current typical MBone data and DPSS data reduces to the concrete problem of corresponding small RTP and RTCP packets with much larger DPSS data blocks. The solution is fortunately quite straightforward: a DPSS data block should contain dozens or hundreds of RTP packets. During recording, the DMRP buffers packets until they constitute a contiguous block that is large enough for efficient storage on the DPSS; during playback, the DMRP obtains data from the DPSS in blocks, but emits the data packet by packet.

The client interface to the DPSS—specifically, the client-side API—permits the client to issue a request for data, and separately to read the data. By always requesting the next data block needed before processing the most recently received block, the DMRP ensures that as much as possible of the latency introduced by the DPSS is subsumed within the time spent processing the prior data block. In the best case, the DPSS can completely process the request and transfer the block over the network while the DMRP is processing the prior block, limiting the client-side delay to the time needed to transfer the data block from

kernel memory to user memory.

III.3 Other Design Considerations

The ability to save entire RTP sessions naturally raises the issue of how the sessions are organized for retrieval. The DMRP does not extensively address this question, but it has a rudimentary built-in naming facility that uses the limited data set management functionality provided by the DPSS itself.

The DPSS development team has been working with the developers of a WWW-based data object manager, WALDO [19], to enable WALDO to catalog all types of data that can be stored on the DPSS. In this way, WALDO will provide DPSS users with a consistent interface by which they can access their data.

WALDO's designers envision allowing the user to browse a collection of data sets (called *large data objects*) via a display of associated information (metadata) pertaining to each; such information could include the data type, owner, size, and a thumbnail image if available. Each entry would also include a hyperlink; by following the hyperlink, the user would cause WALDO to launch an application appropriate for accessing that type of data. In the case of MBone data stored on the DPSS, WALDO would actually launch a DMRP plus whatever other applications were appropriate. For recording it is conceivable that the DMRP would be the only necessary software, but for playback, WALDO would have to

launch end-user tools as well, e.g., vic and vat.

Thus envisioned as "middleware," the DMRP was not given a GUI, and its current model of interaction with the user is quite primitive. Whether simpler and more intuitive means of controlling the DMRP should be built into WALDO or into the DMRP itself is under discussion.

III.4 Design

The DMRP consists of two modules, the *recorder* and the *player*. The recorder and player are each composed of one or more *session manager* modules. Each session manager controls the application's participation in a single RTP session; recording a conference consisting of an audio session and a video session would require two session managers.

Each session manager is comprised of a *packet manager* module and a *DPSS I/O* module. The two modules interact via a shared data buffer under the control of the session manager. In the recorder, the packet manager fills the data buffer and the DPSS I/O module "drains" it by dispatching the buffer's contents to the DPSS as a data block. In the player, the roles are reversed: the

DPSS I/O module "produces" and the packet manager "consumes."

The conceptual design of the DMRP is illustrated in Figure 7:

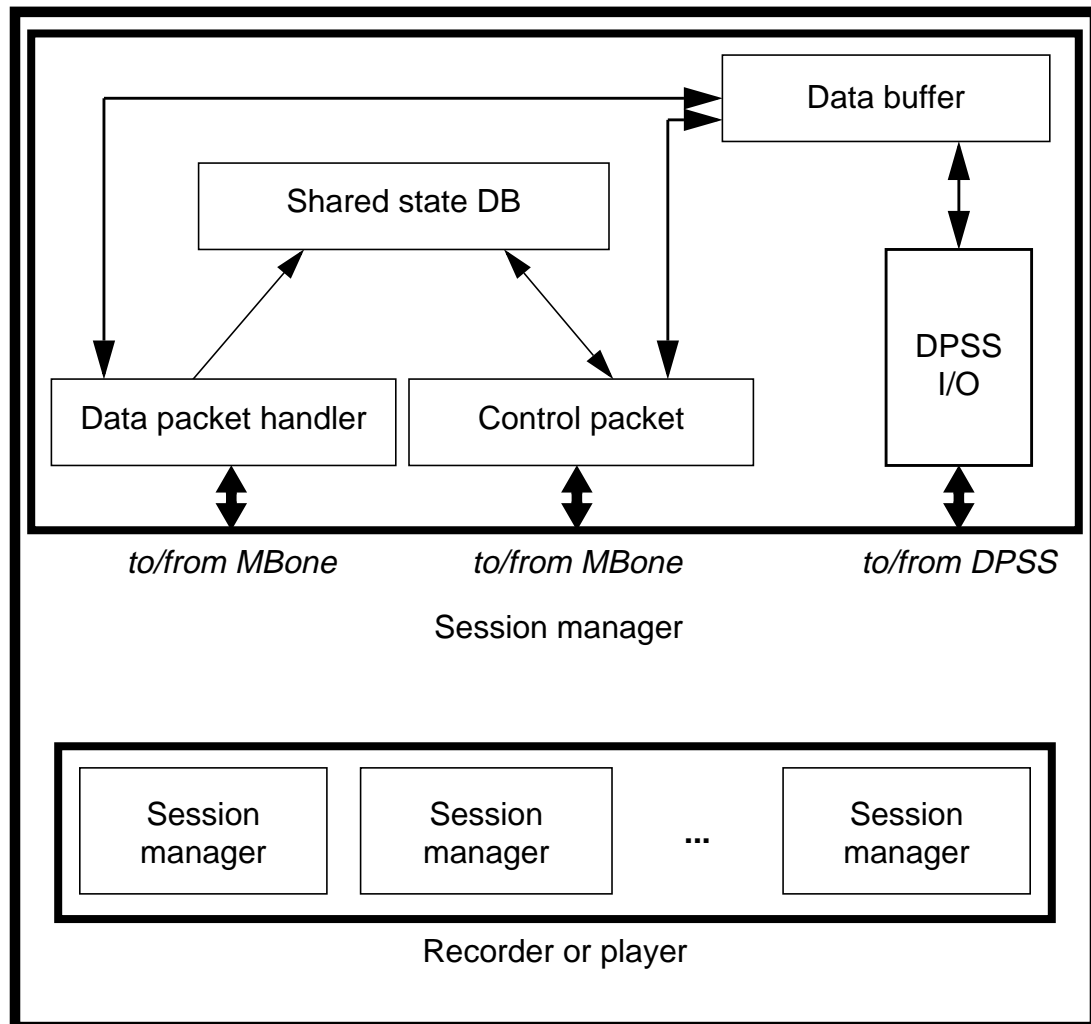


Figure 7. DMRP design

As in Figure 6, arrows indicate the possible directions of data flow; the thicker arrows denote the flow of data to or from the network, with the particular source or sink below the arrow in italics. Note that in the player, the data packet handler

still updates the shared state database, even though the data come from the DPSS via the shared data buffer. This is necessary to ensure that the control packet handler generates RTCP packets that reflect the DMRP's playout rate so that receivers obtain a true picture of the DMRP's performance.

CHAPTER IV

IMPLEMENTATION AND ARCHITECTURE OF THE DMRP

IV.1 Introduction

The DMRP's design decomposed its functionality into relatively independent modules, some of which further decomposed into smaller functional modules. Such modules could have been implemented in many ways and in many languages, but the encapsulation of functionality and state implicit in, e.g., the packet handler, made it natural to view the project in terms of interacting objects. At the same time, it made sense to leverage the functionality offered by the existing DPSS client library (actually a set of several libraries) that offered C language functions to facilitate client interaction with the DPSS. Hence from the beginning, C++ was a logical choice for the language in which to implement the DMRP. An unexpected benefit of coding in C++ was the availability of the C++ Standard Template Library to address the issue of how best to store data that had to be accessed by SSRC ID.

As indicated in Figure 7, both the recorder and player were designed to handle multiple RTP sessions simultaneously, as will usually be necessary for

video conferences (recall that each medium usually constitutes its own RTP session). Each session can largely be handled independently of any other session, since there is no need for session managers to exchange information with one another. Because the session managers' activities are parallel, concurrent, and non-interacting, the session managers were ideal candidates for implementation via threads.

IV.1.1 C++ Terminology and Notation

The description of the DMRP's implementation will occasionally resort to C++-specific notation or terminology, some of which will be reviewed here. A C++ *class* is a type that defines an aggregate of "objects of various types . . . , a set of functions for manipulating these objects . . . , and a set of restrictions on the access to these objects and functions" ([37], 487). The objects and functions within a class are its *members*; member functions are frequently termed *methods*. The C++ :: operator particularizes the scope of a member; subsequent sections of this document will frequently refer to a function *func* that is a member of class *C* as *C::func()*, especially if *func* is referenced in the discussion of a class other than *C*.

An instance of a class is called an *object*. During object instantiation, C++ implicitly invokes a special method of the class called a *constructor*; the constructor allows a class' implementor to ensure that the object is initialized to a

known state. Object instantiation is frequently termed *construction*. A *destructor* is a method that performs any required actions to ensure that the object is in a known state prior to the deallocation of its memory. Not all classes require constructors and destructors, and some classes may have more than one constructor method.

C++ objects control access to their members using the keywords *private*, *protected*, and *public* in the class definitions. These keywords define quite specific conditions under which members are visible or not visible outside the scope of the object. For the purposes of this discussion, however, a private member of object *O* can be considered visible only to, and usable only by, members of *O*, whereas a public member of *O* is visible to and usable by any object or function that can access *O*. In this paper, all methods discussed are public unless stated otherwise.

IV.1.2 C++ Standard Template Library

From the outset, it was clear that the packet manager would need an efficient mechanism to look up per-source data in the RTP/RTCP shared database using a source's (randomly chosen) SSRC ID as a key. Because the DMRP was being implemented in C++, it was possible to take advantage of the recently standardized *Standard Template Library* (STL) [36] to manage the key/data correspondence. STL provides a framework for the use of generalized

algorithms and data types.

A C++ *template*, or *class template*, "specifies how individual classes can be constructed much as a class declaration specifies how individual objects can be constructed" ([37], 595). STL uses templates to parameterize its built-in functions during compilation. The library also relies on the ability of C++ objects to define their behavior when operators such as less-than (<) are applied to them. Thus STL's sorting algorithms, for example, may generically apply < to objects being sorted without regard to the objects' complexity, relying on the objects' implementor to supply a meaningful definition of < as applied to the objects.

IV.1.3 Threads

For code portability, the DMRP adheres to the threads API defined by the formal standard POSIX 1003.1c-1995 [3], approved by the IEEE in June 1995 and hereinafter referred to as "the POSIX threads standard" or simply "POSIX."

Under the POSIX threads standard, a process consists of a single thread executing the main body of the program; this thread is referred to as the *initial thread*. The initial thread differs from all other threads in the process in that if the initial thread is destroyed, all other running threads in the process are also terminated: the entire process is destroyed. The initial thread is also the only one

not explicitly created by the process.

A POSIX thread is created via the `pthread_create()` function, which assigns the new thread an identifier and causes it to execute a function defined by the caller, the *start function*. The newly created thread returns when it has finished executing the start function, but unless specifically allowed to do so, it does not give up its resources until another thread has explicitly joined with it using `pthread_join()`. `pthread_join()` acts as a thread synchronizer by allowing the creating thread to obtain the created thread's return status before the latter is destroyed.

In the DMRP, the initial thread creates a thread to control each of the session managers. The initial thread can control the execution of each of the session managers within the process by setting a status flag in the session manager. Because the flag is accessed by more than one thread—in fact, by two, the initial thread and the session manager thread—inconsistent results are possible if the threads' access is not controlled via mutex locks. POSIX provides mutexes and defines the functions `pthread_mutex_lock()` and `pthread_mutex_unlock()` to acquire and to release mutexes, respectively. The status flag and associated mutex are encapsulated in the `AppStatus` class; see Section IV.2.7 for more details.

IV.2 DMRP Classes

The DMRP recorder and player are implemented as distinct applications. This is a result of the project's development history: it was necessary to develop the recorder first in order to understand the structure of the data the player would be manipulating. However, the two applications must perform some of the same tasks, such as joining RTP/RTCP sessions and both sending and receiving RTCP packets.

The actual implementation of the DMRP has resulted in the two applications sharing most of the same code base in the form of classes which operate one way during recording and the opposite way during playback. The shared code base reflects the two applications' opposite yet highly correlated activities.

The DMRP's class hierarchy, whether recording or playing, can best be understood by considering the following functional areas: generic networking, RTP/RTCP state management, RTP/RTCP packet handling (i.e., filtering), DPSS input and output, per-session management (comprising the interactions between RTP/RTCP packet management and DPSS I/O), and application control. Each functional area is represented by a class hierarchy. In addition, a separate class governs the interaction between the protocol state and protocol packet-handling classes.

IV.2.1 Generic Networking Classes

IV.2.1.1 IPNetAP

At heart, the DMRP's networking needs are quite basic, comprising `open()`, `read()`, `write()`, and `close()`, to use the Unix I/O model.¹¹ The *IPNetAP* class encapsulates the required functionality and hides the specifics of the operating system network programming interface.

The IPNetAP class is oriented to the needs of multicast communication. It contains both a local address (i.e., the local interface through which the session's traffic will be sent and received) and a remote address, usually an IP multicast address and port. After object construction, the application can join a multicast session by invoking the IPNetAP `mcast_connect()` method. `mcast_connect()` requests the operating system to allocate the necessary resources for IP multicast communication and to register the application's desire to join the designated multicast session with the multicast router. (`mcast_connect()` also supports unreliable unicast communication, in which case the remote address is that of a single host rather than a multicast group, and the "connect" behavior is slightly different.)

Following a successful call to `mcast_connect()`, the application can read from or write to the network. `IPNetAP::recv()` and `IPNetAP::recvfrom()` are modeled

¹¹In fact, IP multicast does not require the functionality of `close()` as such, although both RTP/RTCP and Unix sockets do.

on the Berkeley sockets routines of the same name that read data from the network—`recvfrom()`, like its sockets namesake, also returns the address of the host whence the data originated. Similarly, the `send()` and `sendto()` methods transmit data to the networks, `sendto()` allowing the application to specify the address to which the data are to be sent (`send()` uses the remote address specified either during object construction or in the call to `mcast_connect()`).

The `disconnect()` method advises the operating system that the application is finished communicating, allowing the system to recover local resources, and partially resets the `IPNetAP` to indicate that the application may no longer read or write via the object. However, this method does not change either the local or remote address previously set.¹²

The `IPNetAP` class is implemented using Berkeley sockets; several class methods are simply wrappers for sockets interfaces, as evidenced above. One unfortunate implementation flaw is the lack of any method analogous to `select()`: this results in a few instances of other classes having to operate directly at the socket level, violating the encapsulation of the implementation that the `IPNetAP` class otherwise provides. This flaw could be repaired by providing a new class that allows operations across multiple `IPNetAP` objects.

¹²By default, an `IPNetAP` object chooses an address to serve as the local communication endpoint. The application may override this default, as may be desirable for a multihomed host.

IV.2.1.2 IPAddr

The Berkeley sockets API uses the `sockaddr_in` structure to pass IP addresses and transport-level port numbers between many of its routines. The DMRP hides the complications of manipulating `sockaddr_in` structures within the *IPAddr* class. This class provides a number of ways to initialize its value, including methods that translate an ASCII string (representing an IP address in dotted-decimal format) and a 16-bit integer (the port number) to an appropriate `sockaddr_in` for use within the DMRP. The class also provides methods for assigning the value of one *IPAddr* to another and for testing two *IPAddr*s for equality (defined as the same IP address and port number).

The *IPNetAP* class represents its local and remote addresses as *IPAddr* objects. Other classes use *IPAddr* objects whenever address information must be exchanged or manipulated; in particular, the *RTPSourceList* method `get_source()` tests two *IPAddr* objects for equality in order to detect SSRC ID collision.

IV.2.2 RTP/RTCP State Management Classes

IV.2.2.1 RTPSource

RTP requires that each participant use RTP and RTCP packets to monitor and to cache some of the state of every other participant, including each source's SSRC ID, host address, and last sequence number seen. Most of the cached

items are required for RTCP sender and receiver reports, but the SSRC ID and host address are required to detect SSRC ID collisions (two or more session participants accidentally selecting the same ID).

The DMRP encapsulates each session participant's state information, including its own, in the *RTPSource* class. The RTPSource is at the heart of the DMRP's shared-state database, and as such is nothing more than a repository: it performs no processing of the data it stores, leaving that activity to other classes. Its primary roles are to eliminate duplication of data in the packet handler modules and to provide minimal access control to the data. The latter is accomplished via C++ class-based scope control and results in only a limited number of other classes' objects having the ability to modify an RTPSource's contents.

In addition to the SSRC ID, source address, and last sequence number, an RTPSource includes the number of complete cycles through the sequence number space (i.e., how often the source has wrapped sequence numbers), the transit time of the most recent packet, an estimate of the current network jitter, a flag indicating whether the DMRP has received data from the source recently, a flag to indicate whether the DMRP itself has sent data (set only in the RTPSource corresponding to the DMRP itself), the total number of packets and octets sent by the source to date, and several fields used for timing synchronization. There are also class members used to validate a source from

which the DMRP has not previously heard; see the description of the RTPDataHandler, below, for more information on this activity.

The RTPSource was based on a sample data structure provided by RFC 1889, extended to accommodate the DMRP's specific needs (e.g., for timing and synchronization data).

IV.2.2.2 RTPSourceList

Each RTP and RTCP packet's header must be parsed for information either to update or to compare against the data in the appropriate RTPSource, so every packet's arrival requires locating the RTPSource corresponding to the SSRC ID in the packet's header. The *RTPSourceList* encapsulates a group of RTPSource objects and the functionality required to access them.

The most important member function is `get_source()`, which requires an SSRC ID and, optionally, the source's address (if provided, it allows the routine to check for SSRC ID collisions) and returns an RTPSource object. Internally, `get_source()` first attempts to find an RTPSource for the given SSRC ID, and returns the object if it exists (and if there is no SSRC ID collision, if the caller provided a source address); otherwise, `get_source()` builds a new RTPSource object, adds it to the RTPSourceList's set of RTPSource objects, and returns the newly constructed object. Access, therefore, implies creation.

The RTPSourceList's internal database of RTPSource objects is implemented

using an STL associative container called a *map*. An STL map corresponds key values and data values; each unique key is associated with one and only one datum. In this case, each 32-bit SSRC ID is the key that recovers the associated RTPSource object.

IV.2.2.3 RTPSession

The *RTPSession* class is little more than a wrapper for an RTPSourceList and a repository for two per-session attributes, the session's requested bandwidth, and a seed value for a random number generator. The bandwidth (in bytes per second) is needed to calculate the correct interval between RTCP packets generated by the DMRP; a correct packet interval helps to ensure that RTCP does not exceed the fraction of the session's total bandwidth assigned under the application's profile. (The DMRP operates under the general A/V profile described in RFC 1890.)¹³ The seed value is used to prime the random number generator that chooses the DMRP's SSRC ID for the session.

The RTPSession class is a remnant of an earlier stage of the DMRP's implementation. Originally, the class directly implemented RTPSource lookup. However, as the DMRP became more sophisticated, it became necessary to isolate the lookup functionality for use by other classes. The RTPSession will

¹³What the bandwidth for a session should be is not addressed by RTP/RTCP. Some MBone tools, e.g., sdr, allow the user to set the bandwidth and recommend defaults for certain types of sessions.

probably be merged with the only class that derives from it, the RTPComm (see Section IV.2.4), in a future version of the DMRP.

IV.2.3 RTP Packet-Handling Classes

The bulk of the code implementing packet-handling—i.e., parsing of headers (and in the case of RTCP packets, the contents as well), calculating statistics like network jitter, and updating RTPSources with current packet information—is contained in the RTPDataHandler and RTPCtrlHandler classes. However, some common functionality was abstracted into separate classes for clarity and ease of maintenance.

IV.2.3.1 RTPFormat

The *RTPFormat* class contains payload format-specific data. The interpretation of the RTP timestamp's value—or, more to the point, the interpretation of the RTP clock rate—is dependent upon the characteristics of the data being transported. As an example, RFC 1890 mandates that the RTP clock rate for audio encodings "equals the number of sampling periods per second," so that for a typical audio encoding like vat's default, PCM, the RTP timestamp value is incremented by 8000 RTP timestamp units, or *ticks*, per second.

The two most important methods for this class are ticks2ms() and ms2ticks(). ticks2ms() calculates the number of milliseconds corresponding to a given number of RTP ticks for a particular payload format. ms2ticks() calculates the

number of RTP ticks corresponding to a given number of milliseconds for a particular data format; this method is important for generating the RTP timestamp for RTCP SR packets.

The RTPFormat class isolates media-dependent information in a single module so that the rest of the packet-handling classes can be as general and flexible as possible. The RTPDataHandler and RTPCtrlHandler classes inherit from RTPFormat so that they automatically include the functionality needed to interpret RTP timestamp values for both RTCP reporting and RTP data playback.

IV.2.3.2 Timer

The *Timer* class provides routines that monitor inter-packet intervals. Its functionality is important for both the recorder's and player's RTCP packet emissions as well as the player's transmission of data. Each Timer object keeps track of the wallclock time the prior packet was sent, the wallclock time at which the next packet is to be dispatched, and the time remaining until next-packet dispatch.

Four methods are at the heart of the Timer class. `set_send_timer()` takes an amount of time, in milliseconds, and sets the next-packet dispatch time to the session's start time plus that number of milliseconds. `time_to_next_send()` returns the difference between the current time and next-packet dispatch time. `ready_to_send()` returns a nonzero value if the next-packet dispatch time has

been reached or passed, and zero otherwise. Finally, `send_wait()` is used by the `RTPDataHandler` and `RTPCtrlHandler` objects to wait the (usually short) interval until the next packet is to be sent on playback.

The `Timer` class is implemented using the Unix `timeval` structure and relies heavily on the C library's `gettimeofday()` function. It does not, however, rely on traditional Unix timing mechanisms such as those used by `alarm()` or `setitimer()`: these calls rely on `SIGALRM`, which is reserved for use at the application level (see Section IV.3).

IV.2.3.3 RTPHandler

Both the RTP and RTCP packet handlers need to connect to multicast sessions and to disconnect from those sessions; they also need to keep track of the state associated with their presence in the session(s), which may actually represent several different points of contact to the network. The *RTPHandler* class embodies the foregoing functionality.

The `RTPHandler` sets the TTL of the application's packets, governing the packets' effective scope. The `RTPHandler` otherwise consists mostly of wrapper methods for the `IPNetAP` class, with one important distinction: the `RTPHandler` is designed to manage multiple `IPNetAP` objects. This capability is critical for the player, allowing it to simulate the presence of multiple data sources, as may have

existed in the original session.

However, many MBone sessions include many receivers but only one or a few senders. The recorder stores all the RTCP packets that it receives (except as noted in Section IV.2.6), so it would be possible to recreate all receivers' presence on playback; however, doing so would have little practical benefit, for passive receivers contribute nothing to the substance of the playback. More importantly, each recreated session member would require finite system resources such as UDP ports and Unix file/socket descriptors which would be tied up to no good purpose, even if there were enough such resources to do so.

Therefore, the RTPHandler maintains a list of IPNetAP objects, indexed by SSRC ID, corresponding to original session members that actually sent data. An IPNetAP object for a source is created and added to the list only after at least one data packet from that source is encountered in the playback data stream. Thus each member of the original session appears during playback only if it sent data, and only from the moment it first sent data.

IV.2.3.4 RTPDataHandler

The *RTPDataHandler* class encapsulates all the functionality needed for the DMRP to receive and retransmit RTP data packets. Because the DMRP was not intended to modify the data in any way, the RTPDataHandler actually performs all of its limited work upon the RTP fixed header. It inherits from RTPHandler for

packet transmission and receipt functionality, from Timer for inter-packet timing facilities, and from RTPFormat to aid in RTP timestamp interpretation.

The class' main routine for parsing incoming packets is `parse_pkt()`, which is invoked whenever a new RTP packet arrives during recording. The method finds or creates an RTPSource object in the RTPSession based on the RTP fixed packet header's SSRC ID field, then checks the packet header's sequence number and modifies the running estimate of the interarrival jitter. The sequence number check is performed by the private method `update_seq()`, which compares the current packet's sequence number against the last sequence number seen from the source, if any. `update_seq()` requires that a source be considered "on probation" until the RTPDataHandler has received at least two packets with adjacent, in-order sequence numbers. The method also checks both for sequence number wrap, maintaining a counter of such sequence number wraps, as well as for source restart, indicated by a break in sequence number continuity followed by the establishment of a new run of contiguous sequence numbers. `update_seq()` is based on the algorithm provided in Appendix A.1 of [34].

Following sequence number evaluation, `parse_pkt()` updates the current estimate of interarrival jitter, defined as "the statistical variance of the data [packet] interarrival time" ([34], 71), using the private method `update_jitter()`. `update_jitter()` first calculates the raw difference between the current packet's arrival time and its sample time as given in the RTP timestamp field of the fixed

packet header; this difference is the packet's transit time. Then the difference between the current and previous packets' transit times is calculated. The current jitter estimate is subtracted from the difference in transit times, a gain factor of $1/16$ is applied, and the result added to the current jitter estimate. (The gain factor is for noise reduction in the calculations.) `update_jitter()` is based on an algorithm provided in [34], Appendix A.8.

On playback, the method `make_pkt()` is invoked to refashion each saved packet's RTP fixed header as necessary. As noted in Section II.1.2.2, for additional security the RTP timestamp and sequence number should be randomly initialized to new values at the start of playback, with all subsequent replayed packets incrementing from the new values; the SSRC ID should also be randomly initialized to avoid collisions (although collisions are unlikely unless playback occurs in the same session recorded earlier). However, for ease of implementation, the current implementation of `make_pkt()` does not change the original values.

`make_pkt()` updates, or creates, the RTPSource associated with the saved packet's SSRC ID, calling `update_seq()` in the process. It also establishes the correct transmission time by calculating the packet's offset from the first data packet received in terms of RTP ticks. This tick interval is then converted to milliseconds by calling `RTPFormat::ticks2ms()` and this millisecond offset is passed to `Timer::set_send_timer()`. The first data packet received in the original

session, in other words, is treated as the start of that session for timing purposes.

Note that an `RTPDataHandler` object does not independently receive or send packets, that is, the object is not associated with an independent thread of control during recording or playback. Rather, the object is invoked as needed by the thread associated with each `SessionManager`. See the discussion of the `SessionManager` for a more detailed explanation.

IV.2.3.5 RTPCtrlHandler

The *RTPCtrlHandler* class encapsulates all the functionality needed for the DMRP to send and to receive RTCP packets. It parses RTCP packets from other session members, modifying the `RTPSession` database as new members join and as existing members report their current state (e.g., the amount of data they have received from each source, their estimate of the interarrival jitter, etc.). It constructs and transmits the DMRP's own RTCP packets and calculates the interval between each such packet. (Like the `RTPDataHandler`, however, the `RTPCtrlHandler` does not execute within its own thread of control.) The `RTPCtrlHandler` inherits from the `RTPHandler` class for packet transport functionality, from the `Timer` class for inter-packet timing facilities, and from the `RTPFormat` class to aid in RTP timestamp interpretation and generation.

The `RTPCtrlHandler` can be understood by examining five functional areas: construction, parsing of received packets, calculating the pause between

transmitted packets, building packets, and use of stored RTCP packets during playback.

An RTPCtrlHandler object's constructor attempts to initialize the application's RTCP SDES fields as fully as possible based on the user invoking the application, making use of GCOS information in the system's password database or, if available, the .RTPdefaults file created by vic or vat in the user's home directory. The constructor also seeds the random number generator used to create the application's SSRC ID. The constructor requires the name of the program being run (for the SDES TOOL value) and the session's TTL.

After reading an RTCP packet from the network, control passes to the method `parse_pkt()`. `parse_pkt()` first checks the packet's gross structure to verify that some of the fixed fields—e.g., the protocol version and length—are correct, then updates the running average of RTCP packet size and applies packet type-specific methods to process the packet's contents. Only four type-specific methods are currently defined: `parse_sr()`, `parse_rr()`, `parse_sdes()`, and `parse_bye()`, corresponding to the four most common RTCP packet types (see Section II.1.2.3). `parse_sr()` and `parse_sdes()` update the appropriate RTPSource according to the packet's contents. `parse_rr()` only registers a session member's presence in the RTPSession database; further processing is unnecessary since RTCP RR statistics are not currently of interest to the DMRP. Similarly, because the departure of session members is not relevant to the

DMRP, `parse_bye()` does nothing. `parse_rr()` and `parse_bye()` exist as placeholders in case they are needed in future.

`parse_pkt()` "walks" through RTCP compound packets, parsing each constituent packet using the appropriate type-specific method. A successfully processed packet's contents are not altered. An error in any constituent packet invalidates the entire compound packet: there is no provision for continued processing. Because RTPv2 is intended to replace older versions of RTP/RTCP, `parse_pkt()` has not been designed to parse non-RTPv2 control packets and treats them as invalid.

As previously mentioned, RTCP is designed to be self-limiting in its bandwidth consumption, taking no more than a small, fixed percentage of an RTP session's total agreed-upon bandwidth. The RTCP traffic rate is controlled largely by the frequency at which packets are transmitted. `set_rtcp_interval()` calculates the time at which the next RTCP packet should be sent, taking into account both the current number of session members and the estimated average RTCP packet size.

`set_rtcp_interval()` first determines whether a partitioning of the available RTCP bandwidth between senders and receivers is required. All sources that have sent data in the last two reporting intervals are collectively allocated a minimum of 25% of the total RTCP bandwidth; if senders were not given a minimum amount, a small number of senders in a large session would be unable

to send reports frequently enough to be useful to the rest of the session participants. A partitioning of the bandwidth would not be necessary if, for example, the session contained a large number of senders relative to the total number of session members. If it is necessary, `set_rtcp_interval()` determines whether the DMRP has sent any data in order to determine whether it will share the senders' bandwidth or that remaining to the other session participants. Then the average packet size is updated with the size of the last RTCP packet sent by the `RTPCtrlHandler`. Finally, the average packet size is multiplied by the number of members sharing the available bandwidth (giving an estimate of the total amount of data these members would send if they all transmitted one RTCP packet of the average size) and this total is divided by the available bandwidth to produce the interval until the caller should send its next RTCP report.

`set_rtcp_interval()` is based on the algorithm provided in Appendix A.7 of [34].

Whenever an RTCP packet (other than an RTCP BYE¹⁴) should be sent, the application invokes the `send_rpt()` method. `send_rpt()` is misleadingly named, for its primary task is to create the RTCP compound packet that will be sent. It builds the packet in a static buffer created during construction of the `RTPCtrlHandler` object.

`send_rpt()` first checks whether the application has sent data: this will never

¹⁴.To send an RTCP BYE packet, the special method `send_bye()` is used.

be true for the recorder and will usually be true for the player. If data has been sent, `send_rpt()` first creates an SR using the private method `make_sr()`; otherwise, it creates an RR for each source that has sent data during the last two RTCP reporting intervals. RFC 1889 requires that an RTCP compound packet always contain an SR or RR as its first component. If no source has sent data, the packet will begin with an empty RR whose reception report count is zero.

Following the initial SR or RR, `send_rpt()` constructs a two-chunk RTCP SDES using the private method `make_sdes()`. `make_sdes()` invokes another private method, `choose_sdes_items()`, to ensure that the first chunk is a CNAME item and the second chunk either a NAME, EMAIL, or TOOL item. This selection of SDES items is suggested by RFC 1889 to allow rapid distribution to the entire multicast group of the most important information, the CNAME, while also allowing eventual delivery of less important but still useful identifying data.

After building the compound packet, `send_rpt()` transmits it and calls `set_rtcp_interval()` to calculate the time at which the next RTCP packet should be constructed and sent.

Note that `send_rpt()` and most of the methods it invokes require an RTPSource that embodies the current state of the reporter. Only one RTPSource represents the recorder, but the player's state may be represented by several RTPSources, each corresponding to a separate data source in the original session. RTCP packets must be generated for each RTPSource for

which the player is responsible.

Although the DMRP stores RTCP packets along with RTP data packets, the RTCP packets cannot blindly be replayed. Many RTCP data items pertain only to the transient state of the original session, such as the number of lost packets reported by a given receiver, and would be at best meaningless (and more likely misleading) in the context of a retransmission. Indeed, the reasons for which RTCP exists mandate that RTCP packets, and the information they contain, be created *ab initio* during a session.

However, it is useful to identify the original members of a session on playback if they appear, and RTCP SDES packets contain precisely the identifying information needed. Therefore, stored RTCP packets are scanned during playback by the `parse_saved_pkt()` method. `parse_saved_pkt()` traverses the RTCP packet, ignoring all SR and RR components, deleting from the RTPSession any source for which a BYE is read, and passing any SDES components to the routine `update_sdes()`. `update_sdes()` uses the original SSRC ID in the SDES packet to perform a table lookup of the source's SDES information as maintained in a structure called an *sdesinfo*. As with other SSRC ID-based lookups in the packet-handling classes, if such a table entry does not exist, one is created.

At present, `update_sdes()` is only concerned with the source's original SDES CNAME and SDES NAME. The DMRP uses the original CNAME without

modification, under the assumption that if the original CNAME was unique in the original session, it will likely be unique in a replay of the session. However, because most MBone end-user tools use the SDES NAME to identify session members in a human-readable fashion, it seemed appropriate to avoid confusion as to the recorded nature of a playback session by appending " (DMRP replay)" to the original SDES NAME during playback.

As noted in the discussion of the RTPHandler, a member of the original session does not appear as part of the playback session unless and until it sends data. It is helpful, however, that as much identifying data as possible be available for such a sender soon after it appears as part of the playback session. The sdesinfo allows the RTPCtrlHandler to accumulate such identifying data from the source's stored RTCP packets even before it is "known" to have sent data.

IV.2.4 RTP/RTCP Protocol Management Class: RTPComm

The *RTPComm* class attempts to encapsulate all of the data and functionality associated with RTP/RTCP packet handling in the DMRP. It represents the DMRP's interface to a single RTP session, coordinating the functioning of the packet-handling classes (in fact, it contains an RTPCtrlHandler object and an RTPDataHandler object) and the state management classes and hiding much of the complexity associated with both. The RTPComm class thus allows a caller to control the RTP/RTCP classes as a single module using relatively simple and

straightforward semantics such as "join," "read data," and "send control." It embodies the conceptual design of Figure 6. The RTPComm class inherits from RTPSession.

Construction and initialization consists of constructing the member RTPCtrlHandler and RTPDataHandler objects and calling RTPCtrlHandler::set_rtcp_interval() to choose a time to send the application's first RTCP packet (thereby registering the application's presence in the session).

Semantically, the class restricts the DMRP in its interaction with the session to joining, receiving a data or control packet, sending a data or control packet, and disconnecting. The class also includes an encapsulation of the packet handlers' timing facilities to provide hints to the DMRP as to when a packet, either data or control, should be sent.

The join functionality is provided by join()¹⁵, which "connects" the DMRP to the RTP session by triggering mcast_connect() for the RTPComm's RTPDataHandler and RTPCtrlHandler (recall that the required IPNetAP object is encapsulated in one of the objects' base classes, RTPHandler; see Section IV.2.3.3 and Section IV.2.1.1 for more details).

Once the application has determined that either an RTP or RTCP packet is

¹⁵There are actually three join() methods, each allowing the caller to describe the session's multicast address in a different format but otherwise providing the same services.

waiting to be read, it invokes the RTPComm methods `read_data()` or `read_ctrl()`, respectively. `read_data()` in turn invokes the RTPDataHandler to read and to parse the RTP packet. If in parsing the packet the RTPDataHandler indicates that an SSRC ID collision has taken place and the RTPComm determines that the collision involves the DMRP's own SSRC ID, `read_data()` chooses a new SSRC ID before returning. `read_ctrl()` similarly invokes the RTPCtrlHandler to read and to parse the RTCP packet. Both methods return the number of bytes of data read, or -1 on error (e.g., upon encountering an unrecognized or malformed packet header).

Both `read_data()` and `read_ctrl()` also prepend a header structure, dubbed a *PktInfo*, to the raw RTP or RTCP packet. The *PktInfo* contains the packet's size in bytes, its type (RTP or RTCP), and its time offset (currently unused), in milliseconds, from the start of the session.¹⁶ The packet size must be saved because it is part of neither the RTP nor RTCP fixed header: it is provided by the underlying transport protocol (UDP or something equivalent). The size that `read_data()` and `read_ctrl()` return includes the size of the prepended *PktInfo*

¹⁶Formerly, the *time offset* was needed for the player to determine the correct time at which to replay each data packet, relative to a canonical starting time for the session which is maintained by the RTPComm. (This mechanism for timing packet playback was borrowed from the MBone VCR, which uses a similar per-packet header structure.) As described in Section IV.2.3.4, however, the current packet-dispatch timing mechanism is format-dependent and uses the RTP timestamp, as is correct.

structure.

Sending RTCP packets while recording is straightforward: whenever the application needs to send a control packet, it invokes the RTPComm method `send_ctrl()`, which in turn calls `RTPCtrlHandler::send_rpt()`. Sending RTCP packets during playback is slightly more complex. Unlike the recorder, which has only one RTCP endpoint, the player may be responsible for multiple RTCP endpoints, each of which represents a different member of the original session. Moreover, the player must send RTCP packets from each endpoint often enough for receivers to consider the source associated with the endpoint to be alive (i.e., its host has neither crashed nor been severed from the network).

For RTCP timing purposes, the player should calculate an RTCP inter-packet interval for each endpoint as if it were associated with an independent session member. At present, this is not possible because neither the `RTPCtrlHandler` nor `RTPComm` maintains a separate timer for each endpoint: only one timer is available to determine the RTCP interval for the entire application. Consequently, `send_ctrl()` uses a round-robin algorithm to cycle through all the endpoints the `RTPCtrlHandler` has created, calling `RTPCtrlHandler::send_rpt()` on only one of them per RTCP interval. This method will not scale to a large number of playback sources, since receivers will perceive an excessively long RTCP interpacket interval for each source.

`send_ctrl()` requires a flag to indicate whether it is being called within the

recorder or the player.

Sending data packets (which occurs only during playback) is complex for a different reason. It is a two-step process because until the saved packet and its PktInfo have been parsed, the player cannot know when the packet should be sent in the playback session. The RTPComm method `make_pkt()` first checks the type as indicated in the PktInfo, then invokes `RTPDataHandler::make_pkt()` if the packet is RTP data. `RTPDataHandler::make_pkt()` automatically registers the sending time for the packet, so assuming there were no processing errors, the player actually transmits the packet at the correct time using the RTPComm method `send_data_pkt()`. If `RTPComm::make_pkt()` determines that it is processing a saved RTCP packet, it calls `RTPCtrlHandler::parse_saved_pkt()`. `parse_saved_pkt()` does not register a time for the saved RTCP packet to be sent since the packet will not be replayed.

The RTPComm method `wait_send_next()` compares the times at which the next RTP and RTCP packets are to be sent, and only waits until the earlier time, while `RTPComm::send_next()` checks what kind of packet is being sent, and invokes the appropriate packet handler object to send it. `send_next()` also triggers the construction of an IPNetAP object in both the RTPDataHandler and RTPCtrlHandler if a data packet is being sent for the first time from a source, as explained in Section IV.2.3.3. `wait_send_next()` and `send_next()` are necessary to avoid the suspension of the thread controlling the session if it encounters long

periods of time during which no session member sends data (pauses) during a recorded session. (The first implementation of the player simply paused until the next data packet was ready for dispatch. This mechanism worked until the player attempted to play back a lecture containing an unusually long pause. While the player waited until the correct time to send the data packet, it failed to send RTCP packets. The failure to indicate its continuing presence in the session by transmitting RTCP packets caused the other session participants' tools to drop the player from their membership lists, and in a widely distributed session might have caused routers to prune the player's subnet from the multicast distribution tree.)

To disconnect from the RTP session, the RTPComm class provides the method `closecomm()`. `closecomm()` ensures that `RTPCtrlHandler::send_bye()` is invoked prior to the actual disconnect of the relevant IPNetAP objects in the `RTPCtrlHandler` and `RTPDataHandler`.

IV.2.5 DPSS Interface Class: DPSSPOC

The *DPSSPOC* class manages I/O to and from the DPSS. It attempts to isolate the rest of the DMRP from the complexities of the current low-level DPSS client API. It frees the DMRP from explicitly having to monitor some of the state associated with the DPSS session and provides a programming interface that

somewhat resembles the Unix filesystem interface.

Before commencing a DPSS session, a client must specify the DPSS master host and the *Data Set Manager* (DSM) host; the DSM provides a simple naming service for the DPSS and its clients. The DPSSPOC constructor allows the DMRP to specify these hosts. If they are not passed as arguments to the constructor, it searches for appropriate environmental variables for the host names or IP addresses. If either the DPSS master or DSM cannot be ascertained, construction fails.

A DPSS session begins with a call to `open_dpss()`. `open_dpss()` contacts the DPSS master, establishing a connection and a context within which all subsequent transactions during this session will take place. The state reflecting this context is embodied in the *IssHandle*, a data structure provided by the low-level DPSS client API. Clients treat the *IssHandle* in the same way applications treat the Unix FILE structure for buffered I/O, i.e., as an opaque handle required by the interface routines.

To write data to the DPSS, an application must reserve as much space as is needed. Space reservation allows greater control of data layout by client applications, which can take expected data access patterns into account when distributing their data during a DPSS writing session. Reserving space on the DPSS in principle requires that the client application choose individual DPSS disks; in practice, clients need only specify server hosts and by default all disks

on those hosts are used. The DPSSPOC method `prep_new_set()` reserves space on the DPSS for the recorder. It also registers the set with the DSM, as clients are expected to do to facilitate monitoring of the DPSS' contents.

`prep_new_set()` relies on the DPSSPOC method `get_scribe_addrs()`, which takes a string consisting of the desired server names separated by commas or spaces and caches the server names for subsequent use by both `prep_new_set()` and `connect_to_scribes()`. `connect_to_scribes()`, called after a successful return from `prep_new_set()`, sets up a connection between the recorder and the scribe processes on the DPSS server hosts that will be writing the data to disk. Following a successful return from `connect_to_scribes()`, the recorder is ready to store data on the DPSS.

The actual storage of data occurs on a block-by-block basis. When the recorder is ready to commit a block to storage, it invokes the method `send_block()`.¹⁷ `send_block()` uses a counter to construct the block's logical name (see Section II.2.2), calls a low-level DPSS API routine that actually transfers the data, and increments the block counter.

The DPSSPOC is constrained to write one block at a time because the underlying DPSS API routine, `issSendBlock()`, is also so constrained.

¹⁷The recorder actually invokes a wrapper routine, `write_dpss()`; this makes the DPSSPOC programming interface more similar to the standard Unix `write()` system call.

issSendBlock() updates a block lookup table used by the DPSS, the *BlockMap*, with several pieces of information that cannot be determined until the block is actually written. The per-block information includes the block's size in bytes and a checksum. Blocks may be of any size, i.e., may contain any number of bytes of data, up to a maximum of 64 KB. The application includes the exact number of bytes of valid data in the block in the call to issSendBlock().

To read data from the DPSS, the DPSSPOC provides the method `open_dpss_r()`. `open_dpss_r()` connects to the DSM to obtain information about the data set,¹⁸ including the number of blocks comprising it, then connects to the DPSS. Connecting to the DPSS to read data is a four-stage process. The client first contacts the DPSS as in `open_dpss()` to create the DPSS session context, then requests that one or more data sets be prepared for reading. The set preparation request causes the DPSS master to verify that the server hosts on which the data set is loaded are all available; if they are, the master sends the client the information needed to establish the data-requesting and data-receiving streams. The third step is for the client actually to establish these connections to the DPSS, and the final step is to synchronize the client and DPSS by

¹⁸. There are actually two `open_dpss_r()` methods, one of which allows the caller to specify the a set's name and the other of which allows the set to be specified by its DPSS set ID.

exchanging a "ready" message.

The data-request stream is the communications channel through which the client sends its block requests to the master, while the data-receiving streams are the connections by which the client receives the requested data directly from the DPSS servers. As mentioned in Section II.2.3, the client sends block requests to the master, which translates each request to a physical location (using the BlockMap) and forwards the location and number of bytes to read to the disk server. The server reads the data from disk, or possibly finds the data already resident in its memory cache as the result of a prior request, then dispatches the data directly to the client. The DPSSPOC provides the method `request_blocks()` to issue block requests; the method requires only the number of blocks to be requested as a parameter. The DPSSPOC method `receive_blocks()` reads blocks from the DPSS servers into a buffer; both the buffer and the number of blocks to be read are provided as parameters to the method.

By requiring that data reading take place on block boundaries, as opposed to allowing Unix-style `read()` calls of arbitrarily large or small numbers of bytes, the DPSSPOC avoids the delays of memory-to-memory copying between an internal buffer and the user-provided buffer. `receive_blocks()` can read data directly from the network into the user-provided buffer.

At the end of a DPSS session, the DPSSPOC method `close_dpss()` is invoked. If data were being loaded, `close_dpss()` carries out final bookkeeping

tasks, including updating the set's DSM entry to reflect the total number of blocks written, and sending a message to the DPSS master to save the set's BlockMap. `close_dpss()` then ends the DPSS session, using low-level DPSS API routines to close connections and clean up state as necessary.

If a partially filled block is pending when the recorder is shut down, the recorder calls `send_block()` to write the partial block prior to calling `close_dpss()`. Otherwise the pending data would be lost, as the DPSSPOC does not manage the data buffer: it is a resource that must be shared between the DPSSPOC and the RTP/RTCP packet handlers, as is discussed in Section IV.2.6.

IV.2.6 DMRP Session Management Class: SessionManager

The *SessionManager* class orchestrates the interaction between the RTP/RTCP packet-handling classes and the DPSS I/O class. A *DMRP session* consists of an RTPComm participating in a single RTP session (see Section II.1.2.2) with an associated DPSSPOC; the two classes exchange data via a shared buffer. Both classes, together with the shared buffer and some session-wide state data, constitute a *SessionManager*.

IV.2.6.1 Construction and Initialization

Constructing and initializing the *SessionManager* largely consists of constructing and initializing the RTPComm object, and initializing the DMRP session state. Much of this state is encapsulated in a data structure called the

DMRPSessionInfo. The *DMRPSessionInfo* includes the session's multicast or unicast address, port, and TTL, as well as the name by which the set will be (or in the case of playback, is) registered with the DSM. For the most part, the *DMRPSessionInfo*'s data can be obtained automatically from sdr session announcements; see Section IV.3.

IV.2.6.2 Recording

To record a session, the recorder first sets up to write to the DPSS, then joins the session, and finally enters a packet-reading and block-writing cycle. The *SessionManager* method *open_dpss()* performs the steps required to write data to the DPSS, i.e., establishing a DPSS session context, reserving space, connecting to the DPSS scribes, and registering the set with the DSM, all as described in Section IV.2.5. The parameters to *open_dpss()* include the filename to be stored on the DSM, the DPSS master host, DSM host (which need not be the same as the DPSS master host, though it frequently is), the set's estimated size, and a string consisting of the names of the DPSS server hosts on which to store the data.

join_session() is a *SessionManager* wrapper for *RPTComm::join()*, with the *SessionManager* providing the session's address in standard dotted-decimal string form and the port as a host byte-ordered short integer. *join_session()* thus creates a pair of communications endpoints explicitly associated with the

application, resulting in the recorder appearing as a full-fledged member of the RTP session, i.e., as one that identifies itself and its current state (packet statistics) via RTCP.

The SessionManager method `mcast_receive()` reads both data and control packets sent to the session, accumulates them in a buffer, saves the data, and triggers the sending of the recorder's own control packets. The implementation uses the Berkeley sockets call `select()` to poll the RTP data socket and RTCP control socket, using as a timeout the time at which the recorder must build and dispatch the next RTCP packet of its own. The `select()` controls the `while()` loop within which `mcast_receive()` does its work: each iteration of the loop occurs because either an RTP data packet arrived, or an RTCP packet arrived, or the time has come to send an RTCP packet. (However, see Section IV.2.7 for another condition governing the loop's execution.)

If a packet has arrived, `mcast_receive()` calls either `RTPComm::read_data()` or `RTPComm::read_ctrl()`, as appropriate. `mcast_receive()` accumulates packets in the SessionManager's built-in buffer simply by maintaining a pointer into the buffer; this *end-of-buffer* (EOB) pointer is incremented by the number of bytes reported as read by the RTPComm reading routines. In case of error, the EOB pointer is not incremented, so that a malformed or unrecognized packet is obliterated by subsequent valid packets.

After each packet has been read, the number of accumulated bytes in the

buffer—equivalent to the offset of the EOB pointer—is checked against a "high-water mark." The high-water mark, 64 KB, simply reflects the limitation on the maximum size of a DPSS block. If the high-water mark has not been reached, `mcast_receive()` increments the EOB pointer by the number of bytes read; otherwise, it calls the SessionManager method `save_block()`. `save_block()` ensures that no more than 64 KB are written to the DPSS using `DPSSPOC::send_block()`. If the amount of data accumulated exceeds 64 KB, `save_block()` first writes all of the data up to the last complete packet within the 64 KB limit, then copies the excess data from the end of the buffer to the beginning and increments `mcast_receive()`'s EOB pointer so new packets will be stored after the remaining unsent packet. `save_block()` thus ensures that each block begins and ends on the boundary of a packet; this makes playback more efficient because it eliminates the need to copy parts of packets that straddle blocks and guarantees that no packet in one block is delayed in its dispatch because part of it is in the subsequent block.

Whether or not a packet was read, `mcast_receive()` checks whether an RTCP packet should be generated using `RTPComm::ctrl_ready()`, which is a wrapper method for `Timer::ready_to_send()`. If an RTCP packet should be generated, `mcast_receive()` calls `RTPComm::send_ctrl()`.

IV.2.6.3 Automatic Suspension of Recording

Occasionally, relatively long intervals pass during which no RTP session member sends data. It is common, for example, that a lecture source will test its equipment by sending small amounts of data before the actual start of the session, then cease transmitting until the formal start time. Multicast sessions from conferences that span several days often do not transmit data continuously, but only at specified times, e.g., to present keynote speeches.

The DMRP recorder does not permit interactive use as, e.g., the MBone VCR does, so even if a human operator is present, he cannot manually "pause" the recording; the recorder either runs forever or is interrupted. Moreover, it is sometimes impractical for a human being to be present. A major reason to record a session, after all, is likely to be one's inability to watch it live.

Since the recorder continues to run during periods of inactivity, i.e., when no data are being transmitted, it automatically detects the absence of data and ceases recording. Such behavior saves space on the storage device, since the recorder would otherwise continue to store RTCP packets, which are transmitted irrespective of whether data are transmitted. `mcast_receive()` uses a counter, `cycles_sans_data`, to count the number of sequential loop iterations without receiving data. By default, if `cycles_sans_data` exceeds a threshold value, the recorder stops saving RTCP packets. Receipt of a data packet decrements `cycles_sans_data` if it is greater than zero. The reason for decrementing the

counter, rather than resetting it directly to zero, is that short bursts of data—e.g., white noise that exceeds the silence suppression threshold on vat—occasionally are sent to the session by accident. These bursts of data are generally meaningless, but if they reset `cycles_sans_data` to zero, they would increase the amount of time the recorder captured unnecessary RTCP packets.

Decrementing the counter requires data sources to transmit a continuous sequence of packets to keep the counter well below the cutoff threshold, i.e., to "buy" time for pauses between data.

The current threshold value, 10, was chosen at random for testing purposes. Tests with the recorder suggest that this value is too low for real sessions, but further testing will be necessary to determine a more appropriate value. It should also be noted that the recording cutoff can be disabled at runtime if desired.

IV.2.6.4 Playback

Playback first requires connecting to the DPSS and DSM. The player can use the same `open_dpss()` method as described above, using the default file size and DPSS server list parameters (the defaults cause the method to assume that the data set already exists); it may also call `open_dpss()` with a DPSS set ID instead of a filename. In either case, the `SessionManager` invokes `DPSSPOC::open_dpss_r()` to set up for retrieving the data from the DPSS.

The player does not "join" the playback session as the recorder joined the

original session. The recorder joins the original session as a full participant, explicitly identifying itself as a recording process. However, the player has no such distinct identity: the replayed data packets, as well as newly generated RTCP packets, must be attributed to the original data sender(s) rather than to the player process. As such, although the player could join the playback session using `join_session()`, it would be meaningless to do so, for the player will never transmit data via the communications endpoints thus created. Instead, the player calls `SessionManager::register_session()` to record the playback session's address and port in the RTPComm. The RTPHandler object creates the necessary communications endpoints (IPNetAPs) automatically during playback, as noted in Section IV.2.3.3.

`mcast_send()` is the playback analogue to `mcast_receive()`. `mcast_send()` reads data from the DPSS and transmits the data, packet by packet (recall that the original RTP and RTCP packets are saved without alteration), until there are no more to send.

`mcast_send()` implements the scheme described in Section III.2, whereby the request for the next DPSS block is issued just before the current block is parsed. The time required by the DPSS to satisfy the request is thus at least partially subsumed within the time needed to process the current block's data. The cycle is primed by an initial call to `DPSSPOC::read_blocks()`, which is a wrapper for `DPSSPOC::request_blocks` and `DPSSPOC::receive_blocks()`, followed by a call

to DPSSPOC::request_blocks(). DPSSPOC::read_blocks() obtains the first block of the data for immediate processing, while DPSSPOC::request_blocks() begins the request-next/process-current/read-next cycle. At present, only one block at a time is requested and processed, although there is no reason why more could not be handled.

Each DPSS block is parsed packet by packet. mcast_send() performs a crude check for data corruption by examining the prepended PktInfo, then passes the packet to RTPComm::make_pkt() for further handling. Assuming no error occurs, make_pkt() returns the packet's size, not including the size of the PktInfo. If the saved packet was RTCP, mcast_send() updates its pointer into the block buffer (the EOB pointer, analogous to the pointer in mcast_receive()) and begins processing the next packet in the block. Otherwise, mcast_send() sets a "data pending" flag to indicate that a data packet is awaiting transmission and calls RTPComm::wait_send_next() to wait until the next packet (saved RTP or newly generated RTCP) should be sent.

Upon return from wait_send_next(), mcast_send() knows only that at least one packet is ready to be sent, but not which type. mcast_send() therefore checks both whether a data packet should be sent and whether an RTCP packet should be built and sent; it is even possible that both conditions will be true. If the most recently processed data packet is dispatched, the EOB pointer is

updated and the "data pending" flag is cleared.

Following packet dispatch, `mcast_send()` loops back to parse the next packet from the DPSS block. If, however, the currently pending data packet was not sent, `mcast_send()` skips the parsing step and proceeds to call `wait_send_next()`. In this way, a long pause in data transmission during the original session is preserved on playback, but the player is still able to dispatch its required periodic RTCP packets.

IV.2.6.5 Shutdown

The SessionManager contains an AppStatus object (see Section IV.2.7) that acts as a status flag for the loop in `mcast_receive()`. `mcast_receive()` runs until either an error occurs or the AppStatus object indicates that `mcast_receive()` should return. Usually the recorder triggers such a return as a result of some event external to `mcast_receive()`, e.g., a user-requested interrupt. However, `mcast_receive()` may return on its own if it discovers that no more space is available on the DPSS. See Section IV.2.7 and Section IV.3 for more details.

`mcast_send()` reads through blocks packet by packet until it encounters an error or until it determines there are no more blocks to read (or until the user interrupts the process, whichever occurs first; see Section IV.4 for more detail). Like `mcast_receive()`, `mcast_send()`'s main packet-processing loop checks the SessionManager's AppStatus object to see if it should break out of the loop and

return. `mcast_send()` itself will change the `AppStatus` object to indicate it is done if it finishes processing all available blocks; this is the normal means by which the routine quits.

Returning from `mcast_receive()` or `mcast_send()` does not end the DMRP session. The recorder or player must still explicitly exit the session, which means it must call the `SessionManager` methods `close_dpss()` and `leave_session()`. `close_dpss()` first flushes any pending data to the DPSS if the `SessionManager` was recording (the data constitute the last block of the data set), then calls `DPSSPOC::close_dpss()` to shut down all connections to the DPSS and DSM. `leave_session()` is a wrapper method for `RTPComm::closecomm()`.

IV.2.7 Application Status Class: AppStatus

In the original implementation of the recorder and player, the `SessionManager`'s functionality resided within the application itself. Because all of the required state was declared global to the application, it was possible for a Unix signal handler to ensure that on receipt of a signal—e.g., `SIGINT`—the application would shut down gracefully.

The `SessionManager` class was developed because the recorder and player had to be able to accommodate multiple concurrent RTP sessions. Each `SessionManager`, operating independently of other `SessionManagers`, clearly was suitable for implementation as a single thread, particularly when executing

`mcast_receive()` or `mcast_send()`. However, the semantics of Unix signal handling in a multithreaded application made it necessary to modify the original signal handling mechanism.

By default, most signals can be delivered to any thread in the process. In theory, every thread could be allowed to handle the signals by installing appropriate handlers. However, for simplicity the DMRP allows only the initial thread to receive signals for the entire process; the initial thread then takes whatever steps are necessary to communicate with all other threads.

In the original non-threaded implementation, `mcast_receive()` could only be exited by the process' receipt of an asynchronous event, i.e., a signal. (`mcast_send()` returned, in the absence of error, when it ran out of data.) To allow controlled, synchronous communication between the initial thread and the SessionManager threads, it was necessary to modify `mcast_receive()` so that it checked for external events, e.g., the request by the initial thread for `mcast_receive()` to quit.

Thus on each loop iteration, `mcast_request()` now checks the condition of the SessionManager's *AppStatus* object. The AppStatus at heart is merely a flag and a small set of accessor methods to simplify the flag's use. However, the class also includes a mutex to ensure that the flag can be accessed and modified consistently by multiple threads. The methods `lock()` and `unlock()`, used by all the other AppStatus methods to guarantee that only a single thread at a time can

read or modify the flag, are merely wrappers for `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

At present, the only `AppStatus` methods used by other DMRP objects are `done()` and `doneq()`. `done()` sets the `AppStatus`' flag to indicate that the application is ready to shut down. `doneq()` checks whether the flag indicates "done," returning 1 if so and 0 otherwise. In addition to checking both for packets to read and for a timeout, `mcast_receive()` calls `doneq()` on each loop iteration, breaking out of the loop if `doneq()` returns 1.

Although the `AppStatus` class is currently only used to allow the application to indicate that it wishes to shut down, the class can in principle be used to convey other asynchronous events in a synchronous fashion. In addition to responding to data from the network and to timeouts, the packet-processing loop could also implement user-requested actions like "pause," "fast forward," "rewind," and so on, increasing the DMRP's range of functionality. The means of posting the asynchronous event are irrelevant to the responses implemented by the `SessionManager` or other classes, so it would be possible to replace the current terminal-based interface, in which signals are the events, with a GUI in which user-requested window events trigger the actions. Alternatively, the recorder and player could monitor the LBL *Conference Bus* (discussed briefly in [26]) for messages from applications like LBL's *confcntl* [30]. The `AppStatus` class includes the methods `get()`, `set()`, and `add()` to retrieve, to (over)write, and to add

to the status flag, respectively, in the expectation that these capabilities will be useful in future for generalized event posting and monitoring.

Because the AppStatus proved so useful in controlling `mcast_receive()`, `mcast_send()` was also modified to incorporate an AppStatus check in its main loop.

IV.3 Recorder

The recorder controls one or more SessionManagers saving RTP/RTCP data to the DPSS. It allows the user to specify certain DMRP session elements (e.g., the amount of space to reserve on the DPSS for each session) via command-line flags and/or a configuration file. It spawns all necessary threads and coordinates

their shutdown.

Figure 8 shows the recorder operating alongside vic, vat, and sdr. The

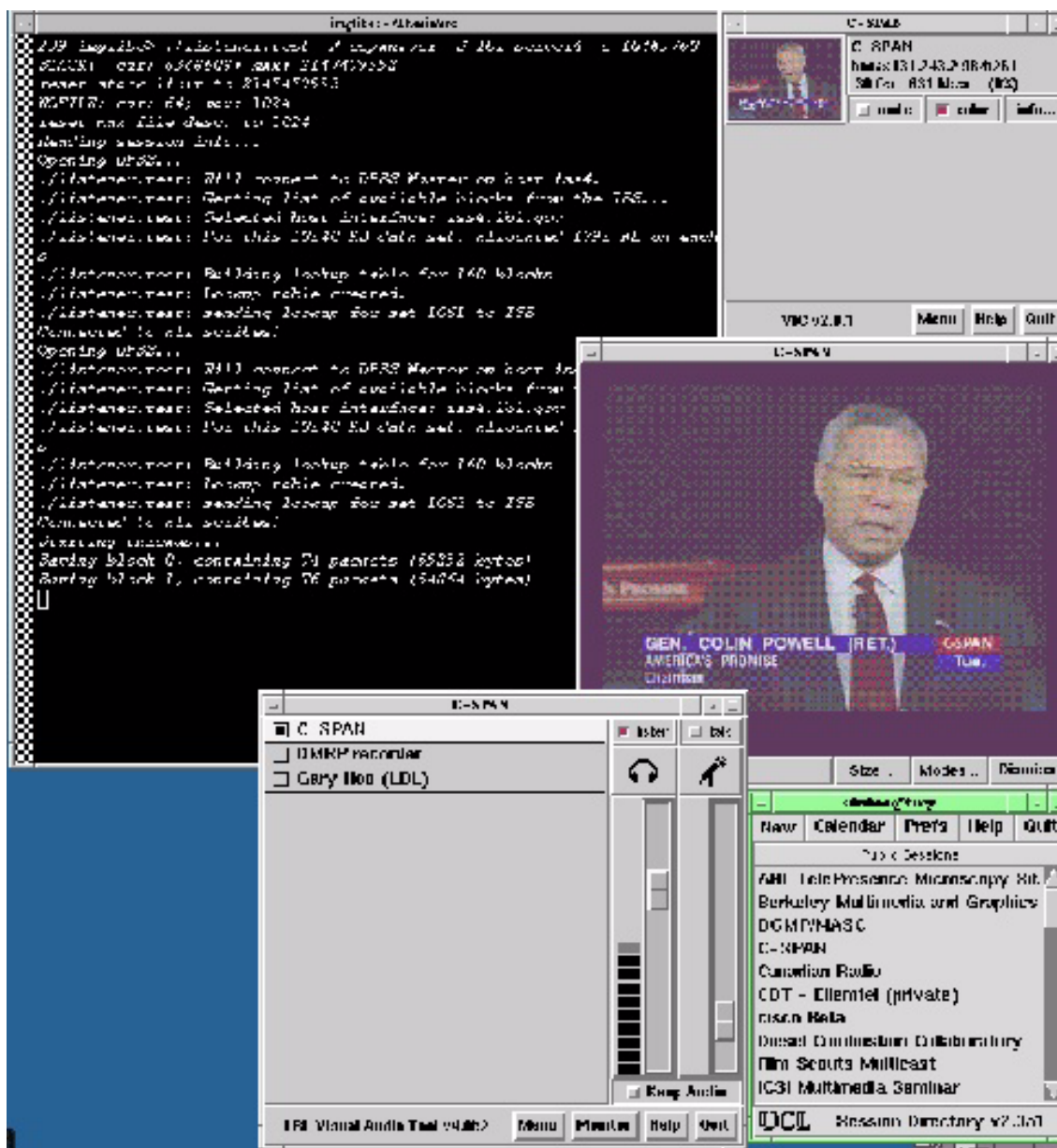


Figure 8. DMRP recorder and other Mbone tools

recorder is capturing the session currently being displayed by vic and vat;

however, vic and vat are operating independently and are not required for the recorder to function. Figure 9 shows the recorder's terminal window from Figure

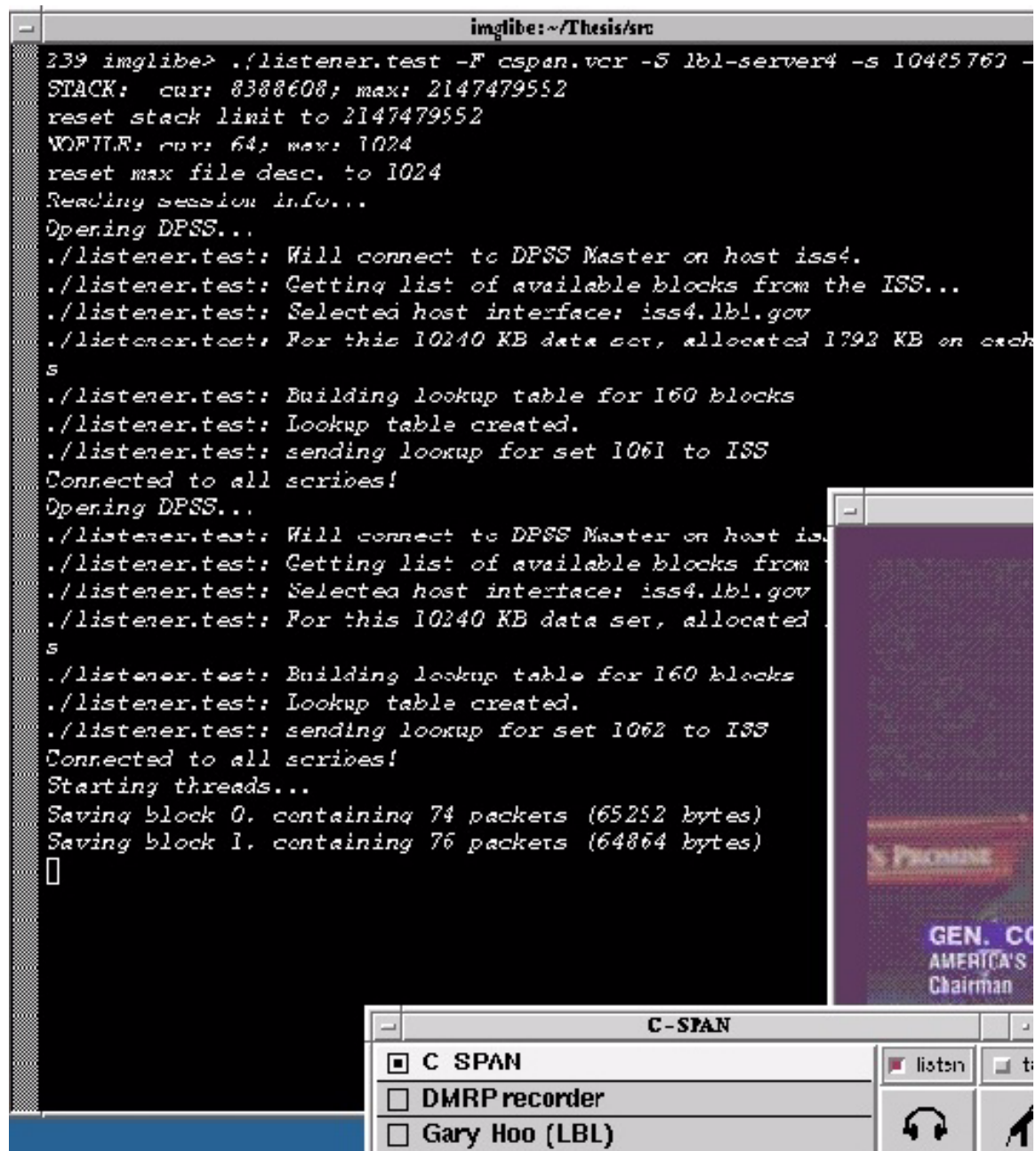


Figure 9. DMRP recorder (detail)

8 in greater detail. (The recorder was invoked as "listener.test" in the figure.)

Recorder initialization, which occurs in the function `init()`, largely consists of argument parsing, followed by `SessionManager` construction and DPSS/DSM connection via `SessionManager::open_dpss()`. Of the fourteen command-line options currently available, four are most often used. The `-F` option allows the user to provide most of the RTP session parameters via an MBone VCR conference configuration file; in Figure 9, the file is called "cspan.vcr".¹⁹ The parameters include the address, port, and TTL to be used for each session (the file can specify more than one). The configuration file's contents are used to initialize `DMRPSessionInfo` structures, each of which is in turn used to construct a `SessionManager` object. Figure 10 shows the `DMRPSessionInfo` structure.

```
struct DMRPSessionInfo
{
    char *addr;
    u_short port;
    int ttl;
    int set_id;
    char *filename;
    u_int filesz;
    u_int bandwidth;
    int duration;
}
```

Figure 10. DMRPSessionInfo structure

¹⁹The MBone VCR file format was chosen for compatibility: Bill Fenner of Xerox PARC wrote a widely available script that can be invoked by `sdr` to generate such a configuration file for an advertised conference.

"addr" is the unicast or multicast address (expressed as a dotted-decimal IP number string) and "port" is the 16-bit port identifier on which the recorder should listen for data. "ttl" denotes the TTL to be associated with the session's packets. "filename" is intended as an aid to human auditors of the DPSS/DSM. "filesz" is the number of bytes to reserve for a session's data, not including RTCP (the recorder automatically allocates a greater number to allow for RTCP). "duration" is the amount of time, in minutes, that the recorder should run. "set_id" is useful only to the player and is discussed in Section IV.4; "bandwidth" is currently unused.

The -S flag is required to list the DPSS server hosts on which the data should be stored. The server list must be parsed as a single argument by the Unix shell, so it must consist either of a comma-separated list without spaces, or a list, possibly space-separated, enclosed by quotation marks. (DPSS clients use the DPSSHOST environmental variable to set the DSM / DPSS master host.)

The -s flag allows the user to specify how many bytes to allocate for each data set (corresponding to an RTP session); the default is 1,073,741,824 bytes, or 1 GB.²⁰ (In Figure 9, the recorder is shown requesting 10,485,760 bytes, or 10 MB.)

In an attempt to automate recording, the recorder allows the user to set a

²⁰This has proven to be somewhat more than is usually necessary for current MBone data rates and session duration.

recording length via the `-l` option; the length is specified in minutes. This eliminates the need for a human user to monitor a session or sessions whose duration is announced in advance, and facilitates automated recording via a tool like WALDO (see Section III.3), which can launch a recorder process without requiring a mechanism to deliver it a signal or otherwise to micromanage its activities.

Following initialization, the recorder calls the function `join_sessions()`, which in turn invokes `SessionManager::join_session()` for each `SessionManager` object. `join_sessions()` also ensures that each `RTPComm` has the same "session start" time, so that packet time offsets are measured from the same starting time across all DMRP sessions. `join_sessions()` returns the number of sessions for which `SessionManager::join_session()` did not return an error. In theory, the recorder could operate as long as at least one `SessionManager` successfully joined its session; at present, however, the recorder quits if any `SessionManager` failed to do so.

After a successful return from `join_sessions()`, the initial thread creates a thread for each `SessionManager`; each such *session thread* immediately begins executing `SessionManager::mcast_receive()`. If the user requested a timeout via the recorder's `-l` option, the initial thread spawns a *timer thread* that issues a call to `alarm()`, which will result in a `SIGALRM` being sent to the process when it has

run for the requested amount of time. Finally, the initial thread waits for signals.

Prior to creating other threads, the initial thread blocks the signals it wishes to catch using the POSIX function `pthread_sigmask()`; in the recorder's case, the signals of interest are `SIGINT` and `SIGALRM`. `pthread_sigmask()` changes the default action taken in response to the specified signals, modifying the thread's signal mask. In the case of the recorder, the signals are blocked, or queued for delivery (as opposed to being delivered asynchronously as is normal). Any new thread inherits its creator's signal mask, so that all threads block these signals, enabling the initial thread to use the system call `sigwait()` to catch the blocked signals synchronously. `sigwait()` waits for one of a set of signals to be posted to the process; the set of signals is passed as a parameter to the call.

A `SIGINT` or `SIGALRM` received by `sigwait()` causes a controlled shutdown. The initial thread calls `AppStatus::done()` for each `SessionManager` via the `SessionManager` wrapper routine `quit()`. Then the initial thread joins each session thread, releasing the resources associated with the thread, like memory. Note that the `SessionManager` objects are not deallocated with the session threads since the initial thread allocated their memory. Finally, the initial thread calls the recorder function `close_session()` for each `SessionManager`. `close_session()` first calls `SessionManager::close_dpss()`, then `SessionManager::leave_session()`.

IV.4 Player

The player controls one or more SessionManagers playing RTP/RTCP data from the DPSS. It allows the user to specify DMRP session elements via command-line flags and/or a configuration file, spawns all necessary threads, and coordinates their shutdown, like the recorder.

The player's operation is shown in Figure 11. The player is running in the

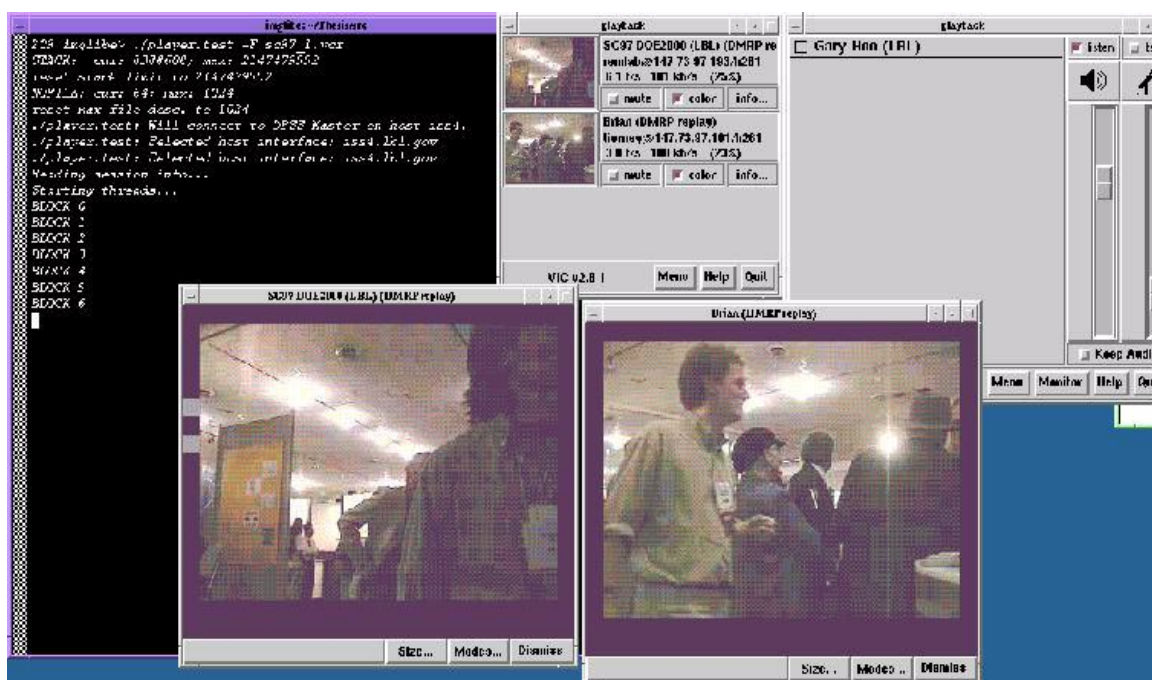
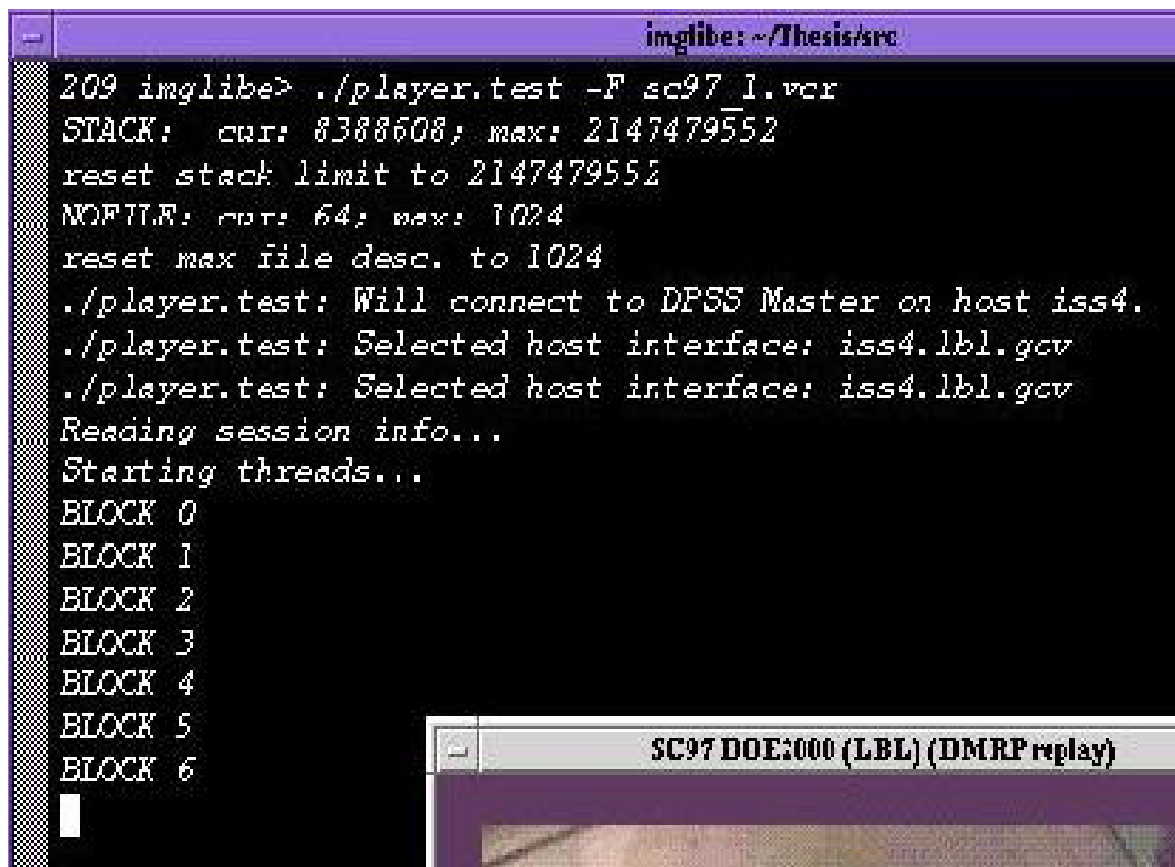


Figure 11. DMRP player and client applications

terminal window on the left of the figure. The session being replayed consists of a single video session with two senders, as indicated by the two vic windows at the bottom center, but no corresponding audio; thus the vat client at the top right corner of Figure 11 shows no other session participants and no audio output.

(The original senders had muted their audio devices due to the noisiness of the environment in which the original conference had been recorded.)

The player's command-line interface is shown in more detail in Figure 12 .



```

imglib: ~/Thesis/src
209 imglib> ./player.test -F sc97_1.vcr
STACK:  cur: 8388608; max: 2147479552
reset stack limit to 2147479552
NOFILE: cur: 64; max: 1024
reset max file desc. to 1024
./player.test: Will connect to DPSS Master on host iss4.
./player.test: Selected host interface: iss4.lbl.gcv
./player.test: Selected host interface: iss4.lbl.gcv
Reading session info...
Starting threads...
BLOCK 0
BLOCK 1
BLOCK 2
BLOCK 3
BLOCK 4
BLOCK 5
BLOCK 6

```

Figure 12. DMRP player (detail)

(The player was invoked as "player.test.") Note that the only command-line option provided was a configuration file via the -F option. Also, by default, each session thread prints the number of each DPSS block as it is being processed; this verbosity helps to diagnose errors such as the player sending to a

different destination address than the client expected. Finally, the title bar of the client vic window visible in Figure 12 indicates that the sender "SC97 DOE2000 (LBL)" is not live, but is rather being replayed by the DMRP.

Player initialization is similar to recorder initialization: the function `init()` parses arguments, constructs `SessionManager` objects (one per session to be replayed), and connects them to the DPSS and DSM via `open_dpss()`. The most important command-line option flag is `-F` for the configuration file, which must be in MBone VCR format. Note that the configuration file must include an address, port, and TTL for each medium to be replayed, and it is the user's responsibility to ensure these do not conflict with existing sessions. `sdr` can be used to choose these parameters by creating a new presentation to be advertised; `sdr` can also generate the configuration file containing these parameters for the player.

The configuration file must also specify what DPSS set(s) contain the data to be replayed. Although the recorder generates a name for each set (the `DMRPSessionInfo` "filename" field mentioned in Section IV.3), that name is not guaranteed to be unique on the DPSS; in fact, it is guaranteed to be the same for multiple recordings (in parallel or in sequence) that use the same configuration file. Moreover, the player requires the original configuration file to reconstruct the names chosen by the recorder. For these reasons, the DMRP extends the MBone VCR configuration file format to allow a DPSS set ID to be designated for each medium: this is used to fill in the `DMRPSessionInfo` "set_id" field. Because

the set ID is guaranteed to be unique on a given DPSS, users are encouraged to identify sets for playback in this way rather than by name.

Following a successful return from `init()`, the player calls `register_sessions()`, which calls `register_session()` for each `SessionManager` and, like the recorder function `join_sessions()`, ensures a uniform "session start" time across all RTPComm objects. Then the player creates a session thread for each `SessionManager`. Each session thread executes `SessionManager::mcast_send()`.

Shutdown is handled somewhat differently from the recorder. Each session thread runs to completion, i.e., until there are no more blocks to read from the DPSS. Following a successful return from `mcast_send()`, the session thread calls the player function `dec_thread_count()`, which decrements a global counter of the number of active session threads. The last thread to complete detects that the counter is zero after it calls `dec_thread_count()` and sends the player process a SIGINT. Because the player, like the recorder, created its session threads with SIGINT blocked, the signal is delivered to the initial thread, which is blocked in `sigwait()`. Following return from `sigwait()` (as a result of signal delivery), the player joins each session thread and invokes `close_session()` for each `SessionManager`.

CHAPTER V

PERFORMANCE AND SCALABILITY

The following discussion considers the performance and scalability of the DPSS and DMRP.

Because the DPSS consists of multiple disks, it inherently has greater total storage capacity than any single disk. Moreover, the DPSS can parallelize disk accesses and network sends across its multiple disks in a way that is impossible to duplicate on any single host with a single disk. This means that the DPSS can take full advantage of network bandwidths that exceed the maximum throughput of any single disk.

For example, a DPSS in current use for testing consists of three server hosts, each of which is a Sun Microsystems UltraSPARC I workstation with an OC-3 ATM interface capable of a raw throughput of 155 Mb/s. Cavanaugh [4] suggests that 135 Mb/s is the maximum data rate in practice after taking link-layer and ATM protocol overhead into account (this does not include TCP and IP protocol overhead, but these can be considered minimal for sufficiently large TCP packets and assuming no fragmentation at the IP level). Attached to each host are two disks, each of which has a storage capacity of 4501 MB (4.4

GB) and a raw throughput measured at approximately 14 MB/s, or 112 Mb/s; note that neither of these disks is sufficient to saturate a server's ATM network interface by itself. In toto, this 3-server configuration is theoretically capable of a sustained transmission rate of approximately 405 Mb/s. Measurements of DPSS throughput capability have demonstrated 80 Mb/s throughput rates per disk server for an aggregate of 240 Mb/s for this configuration. (It is worth noting that this throughput measurement was made with a DPSS client that ignored the data; this figure can therefore be considered an upper limit for what "real" clients can expect to achieve.) Details of this testing are provided in Appendix A.1.

It is useful to compare the DPSS' data rate with currently achievable MBone data rates. For pulse code modulation (PCM)-encoded data [33]—the default audio encoding used by vat—320-byte packets of data are transmitted every 40 ms. The RTP header adds 12 bytes, UDP adds 8, and IP adds 20 for a protocol overhead of 40 bytes (12.5%) per packet, resulting in a raw data rate of 9000 bytes/s, or a useful data rate of 8000 bytes/s (64 Kb/s). vic's default encoding, H.261 [40], can generate up to 30 frames/s, but the frame size varies. For a high-bandwidth video session, an average frame is 2700 bytes (according to observation of typical high-bandwidth video sessions). Because vic typically breaks up frames into three or four RTP packets, 120-160 bytes of each frame are required for network protocol headers. Thus a high-bandwidth video session has a raw data rate of between 84,600 bytes/s (660 Kb/s) and 85,800 bytes/s

(670 Kb/s), and a useful data rate of approximately 81,000 bytes/s (632 Kb/s).

Thus the combined audio and video from a single participant can be expected to require approximately 700 Kb/s of bandwidth. The DPSS described above can in theory sustain approximately 350 such combined audio and video sessions simultaneously (either recording or playing), assuming that the underlying network is capable of providing the bandwidth. Unfortunately it is not currently feasible to undertake such a test in the current LBL environment due to insufficient hosts on which to run the recorders or players.

Tests (described in Appendix A.2) have demonstrated that a single DMRP player can support twelve conferences of combined audio and video, each conference averaging 700 Kb/s in throughput, for a total of 8.2 Mb/s throughput for the application. Furthermore, a DMRP recorder can record twelve such conferences with minimal loss beyond that caused by congestion.

In these tests, a player on a Sun Microsystems Ultra Enterprise Server (with ~1 GB of memory and a 620 Mb/s OC-12 ATM interface) was used to play back three separately recorded presentations (each consisting of one audio and one associated video session with one data source per session) in parallel: that is, the three presentations—or six RTP sessions—were replayed simultaneously. Furthermore, the simultaneous replay was repeated to four new conferences, where each new conference consisted of an audio session and a video session. Each RTP session corresponds to a DPSS data set and each data set to a client,

so the player simulated 24 separate DPSS reading clients at the same time. The recorder (running on a second, similarly configured Ultra Enterprise Server) was used to record the four new conferences. Because each conference consisted of an audio and a video session, the recorder simulated eight separate DPSS writing clients simultaneously. Thus the DPSS was able to serve 32 clients, receiving data at approximately 8 Mb/s while transmitting at the same rate. Playback of the four newly recorded presentations showed that little data had been lost.

CHAPTER VI

FUTURE WORK

The DMRP has much room for improvement. Perhaps its most obvious flaw is its lack of interactivity: there is no way for a user to change its behavior once it has started running. Although originally envisioned as middleware that would not require extensive user control, in practice it has turned out to be similar enough to a hardware audio or video recorder that a GUI should be added to allow pausing, stopping, rewinding, fast-forwarding, and so on. Some modification to the SessionManager and AppStatus classes will be necessary to support this new functionality. Also, as noted in Section IV.2.7, the AppStatus class should be modified to allow remote control, perhaps via RTSP (see Section II.1.3). Also, in order to minimize the difficulty of compiling and installing the GUI-based application, the recorder and player should be combined into a single program.

One of the DMRP's most troublesome shortcomings is its inability to request additional space dynamically from the DPSS. Because the DPSS was originally designed to accommodate read-mostly data sets whose size was known in advance, the API does not provide a simple way to request more space after the initial allocation. However, the code to do so has been written for other

applications and the DPSSPOC class should be modified accordingly.

The DMRP's interaction with the DPSS should be made more efficient by adding a double-buffering system. At present, when the recorder needs to save a block, `mcast_receive()` pauses in its packet processing in order to send the block to the DPSS. Depending on the data rate, several data packets may be lost in this interval. A more efficient mechanism would employ an additional buffer and an additional thread: when one buffer is full (i.e., ready to be stored on the DPSS), the thread executing `mcast_receive()` would begin saving into the other buffer with virtually no delay; meanwhile, the new thread would save the full buffer to the DPSS. A similar problem exists in the player and a similar mechanism should be introduced for reading blocks from the DPSS during playback.

The thread-unsafe nature of the DPSS client library requires that each session thread in the recorder and player establish its own connection to the DPSS and maintain its own state in the form of a distinct `IssHandle` (see Section IV.2.5). Much of this state is redundant: for example, all the session threads will store data to the same DPSS disks or read data from the same server hosts. If the DMRP could eliminate the redundant state, it could reduce its memory footprint. Whether it is actually possible to synchronize access to shared DPSS client library state without corrupting that state is unclear, however.

The DMRP violates RTCP's requirements in a few minor respects. Although

the specification calls for an SSRC's CNAME to be consistent throughout a session, the player may not be able to assign the source's correct CNAME when data playback begins because the player has to wait until it reads a saved RTCP packet from the source in the saved data stream. Hence the CNAME for a source may change, particularly if it sends data at or near the beginning of playback. The violation could be corrected by creating a cache of SSRC identifying information. The cache could be created either during recording, off-line between recording and playback, or just before playback commenced; it could be stored with the data set or off-line as part of WALDO.

Once the RTPCtrlHandler has filled in an SDES NAME for an SSRC on playback, it ignores any changes that may have occurred in the original session (e.g., a user may not have set his NAME correctly in his tool prior to sending data, but may have done so subsequently). Again, caching the SSRC identifying information would correct this playback error, as would modifying the RTPCtrlHandler to notice when such information changes for an SSRC. (Note that showing an incorrect NAME does not violate RTCP, nor does the protocol prohibit modifying the NAME.)

For security, the player should generate new RTP timestamps, sequence numbers, and SSRC IDs for all data packets, as noted in Section IV.2.3.4.

Playback as currently implemented makes the generation of RTCP packets at correct intervals for each playback source difficult, as noted in Section IV.2.3.5.

The RTPCtrlHandler should be modified to support multiple interval timers so that each advertised SSRC appears to send its own RTCP packets at the proper times.

At present, the DMRP recorder does not reorder out-of-order packets before storing them, nor does the player reorder out-of-order packets before (re)transmitting them. Although misordering has not been a significant problem in tests to date, some kind of reordering—perhaps performed off-line between recording and playback—would prevent the player from reproducing the original session's errors and could be important in an environment where misordering occurs more frequently, e.g., on an Ethernet.

As noted in Section IV.2.6.3, automatic recording suspension has not been adequately tested to find the proper number of non-data cycles that should elapse before shutting off packet saving.

During testing with real Mbone sessions, the DMRP recorder showed a lack of robustness: it crashed unexpectedly when an advertised but initially inactive video session suddenly became active, i.e., a participant began sending low-rate video. More testing is needed to determine the cause of the problem.

The RTPFormat class needs to know about more payload formats; at present only H.261 (for video) and PCM (for audio) have been tested extensively, although several other audio formats have been tested for short times. Attempts to test JPEG video were unsuccessful because of problems generating a

sustained, high-bandwidth source stream using vic.

CHAPTER VII

CONCLUSIONS

The DMRP mediates between multicast multimedia traffic on the Internet and a distributed server, the DPSS. The DMRP captures RTPv2 and RTCP traffic from one or more RTP sessions, accumulating data into large blocks and saving these blocks onto the DPSS. On playback, the DMRP reproduces the original sessions' flow of data by reading blocks and parsing them, packet by packet, according to their original timing. During playback, the DMRP also retransmits appropriate identifying information so that data can be attributed to the original sources rather than to the player process.

Whether recording or playing, the DMRP attempts to comply fully with the requirements of RTP and RTCP so that it is a full participant in an RTP session: in particular, the DMRP transmits RTCP packets at appropriate intervals. Protocol compliance is important to allow data sources to monitor and, if necessary, to modify their behavior based on current conditions.

The DMRP's implementation strives to make modifications relatively easy while taking advantage of existing technology. The use of object technology provides a clean organization of, and division between, code modules that

resembles the original design; it also permits the extension of the existing class hierarchy to allow new, unanticipated behavior if it is desirable or necessary. Moreover, the choice of C++ as the implementation language permits the DMRP to take advantage of the C++ Standard Template Library as well as the substantial existing C language DPSS API. Implementing much of the functionality via concurrently running threads allows a modest degree of scalability.

REFERENCES

- [1] Berc, L., W. Fenner, R. Frederick and S. McCanne. 1996. *RTP Payload Format for JPEG-compressed Video*. IETF Request for Comments 2035.
- [2] Braden, R., L. Zhang, S. Berson, S. Herzog and S. Jamin. 1997. *Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification*. IETF Request for Comments 2205.
- [3] Butenhof, D. R. 1997. *Programming with POSIX(R) Threads*. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley.
- [4] Cavanaugh, J. 1994. "Protocol Overhead in IP/ATM Networks." Available from <http://www.msci.magic.net/Papers.html>.
- [5] Chen, L. T. and D. Rotem. 1993. "Declustering Objects for Visualization," *Proc. of the 19th VLDB (Very Large Database) Conference*.
- [6] Deering, Steve. 1989. *Host Extensions for IP Multicasting*. IETF Request for Comments 1112.
- [7] Eriksson, Hans. 1994. "MBONE: The Multicast Backbone," *Communications of the ACM* (August 1994/Vol 37, No. 8): 54-60.
- [8] Fenner, W. 1997. *Internet Group Management Protocol, Version 2*. IETF Internet Draft draft-ietf-idmr-igmp-v2-06.txt (January 1997; work in progress).
- [9] Fielding, R., J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee. 1997. *Hypertext Transfer Protocol -- HTTP/1.1*. IETF Request for Comments 2068.
- [10] Floyd, S., V. Jacobson, C. Liu, S. McCanne and L. Zhang. 1995. "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," *ACM SIGCOMM 95* (August 1995): 342-356.
- [11] Greiman, W., W. E. Johnston, C. McParland, D. Olson, B. Tierney, C. Tull. 1997. "High-Speed Distributed Data Handling for HENP," International Conference on Computing in High Energy Physics (April

1997), available from <http://www-itg.lbl.gov/STAR/>.

- [12] Handley, M. 1996. *SAP: Session Announcement Protocol*. IETF Internet Draft draft-ietf-mmusic-sap-00.ps (November 1996; work in progress).
- [13] Handley, M., J. Crowcroft, C. Bormann and J. Ott. 1997. *The Internet Multimedia Conferencing Architecture*. IETF Internet Draft draft-ietf-mmusic-confarch-00.txt (July 1997; work in progress).
- [14] Handley, M. and V. Jacobson. 1997. *SDP: Session Description Protocol*. IETF Internet Draft draft-ietf-mmusic-sdp-04.ps (September 1997; work in progress).
- [15] Handley, M., H. Schulzrinne and E. Schooler. 1997. *SIP: Session Initiation Protocol*. IETF Internet Draft draft-ietf-mmusic-sip-03.txt (July 1997; work in progress).
- [16] Hoffman, D., G. Fernando and V. Goyal. 1996. *RTP Payload Format for MPEG1/MPEG2 Video*. IETF Request for Comments 2038.
- [17] Holfelder, W. 1995. "Mbone VCR - Video Conference Recording on the MBone," *ACM Multimedia 95* (November 1995).
- [18] Hoo, G. 1996. *tv_sim home page*. Available at http://www-itg.lbl.gov/ISS/userguide/tv_sim.html.
- [19] Johnston, W., G. Jin, C. Larsen, J. Lee, G. Hoo, M. Thompson and B. Tierney. 1997. "Real-Time Digital Libraries based on Widely Distributed, High Performance Management of Large-Data-Objects" (draft submitted to *International Journal of Digital Libraries*, special issue on "Digital Libraries in Medicine," available at <http://www-itg.lbl.gov/WALDO/DigLib/LargeDataObj-Arch.fm.html>).
- [20] Johnston, W. E., B. L. Tierney, H. M. Herzog, G. Hoo, G. Jin and J. R. Lee. 1994. "Distributed Parallel Data Storage Systems: A Scalable Approach to High Speed Image Servers," *ACM Multimedia 1994*.
- [21] Johnston, W. E., B. L. Tierney, H. M. Herzog, G. Hoo, G. Jin and J. R. Lee. 1994. "Using High Speed Networks to Enable Distributed Parallel

Image Server Systems," *Supercomputing '94*.

- [22] Johnston, W., B. Tierney, J. Lee, G. Hoo and M. Thompson. 1996. "Distributed Large Data-Object Environments: End-to-End Performance Analysis of High Speed Distributed Storage Systems in Wide Area ATM Networks," *Fifth NASA/Goddard Conference on Mass Storage Systems and Technologies* (available at <http://www-itg.lbl.gov/%7Ejohnston/DPSS.NASA.MSS.Symp96.9.fm.html>).
- [23] Kouvelas, I. and V. Hardman. 1997. "Overcoming Workstation Scheduling Problems in a Real-Time Audio Tool," *Proceedings of Usenix Annual Technical Conference* (January 1997): 235-242.
- [24] Leclerc, Y. and S. Q. Lau. 1994. "TerraVision: A Terrain Visualization System," Technical Note 540, SRI International, available from <http://www.ai.sri.com/~magic/terravision.html>.
- [25] McCanne, S. 1992. "A Distributed Whiteboard for Network Conferencing" (unpublished term project report, University of California, Berkeley, May 25, 1992; available from <http://www.cs.berkeley.edu/~mccanne/unpublished.html#wb-work>).
- [26] McCanne, S. and V. Jacobson. 1995. "vic: A Flexible Framework for Packet Video," *ACM Multimedia* (November 1995): 511-522.
- [27] Mills, D. 1992. *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. IETF Request for Comments 1305.
- [28] Network Research Group, Lawrence Berkeley National Laboratory. *LBNL Audio Conferencing Tool (vat) home page*. Available at <http://www-nrg.ee.lbl.gov/vat/>.
- [29] Perkins, C., I. Kouvelas, O. Hodson, V. Hardman, M. Handley, J.C. Bolot, A. Vega-Garcia and S. Fosse-Parisis. 1997. RTP Payload for Redundant Audio Data. IETF Request for Comments 2198.
- [30] Perry, M. 1997. *Confcntlr home page*. Available at

<http://www-itg.lbl.gov/mbone/confcntl.r>.

- [31] Postel, Jon. 1981. *Internet Protocol*. IETF Request for Comments 791.
- [32] Postel, Jon. 1982. *Simple Mail Transfer Protocol*. IETF Request for Comments 821.
- [33] Schulzrinne, H. 1996. *RTP Profile for Audio and Video Conferences with Minimal Control*. IETF Request for Comments 1890.
- [34] Schulzrinne, H., S. Casner, R. Frederick and V. Jacobson. 1996. *RTP: A Transport Protocol for Real-Time Applications*. IETF Request for Comments 1889.
- [35] Schulzrinne, H., A. Rao and R. Lanphier. 1997. *Real Time Streaming Protocol (RTSP)*. IETF Internet Draft draft-ietf-mmusic-rtsp-04.txt (September 1997; work in progress).
- [36] Stepanov, A. and M. Lee. 1995. *The Standard Template Library*. Hewlett-Packard. Available from <http://www.cs.rpi.edu/projects/STL/htdocs/stl.html>.
- [37] Stroustrup, B. 1991. *The C++ Programming Language*. Reprinted with corrections 1995. Reading, Mass: Addison-Wesley.
- [38] Thompson, M., W. Johnston, G. Jin, J. Lee, B. Tierney and J. F. Terdiman. 1996. "Distributed Health Care Imaging Information Systems." Available from <http://www-itg.lbl.gov/DPSS/Kaiser/>.
- [39] Tierney, B., W. Johnston, H. Herzog, G. Hoo, G. Jin and J. Lee. 1994. "System Issues in Implementing High Speed Distributed Parallel Storage Systems," *Proceedings of the USENIX Symposium on High Speed Networking*.
- [40] Turetti, T. and C. Huitema. 1996. *RTP Payload Format for H.261 Video Streams*. IETF Request for Comments 2032.
- [41] Wiltzius, D., L. Berc and S. Devadhar. 1996. "BAGNet: Experiences with an ATM metropolitan-area network," *ConneXions* 10, no. 3 (March

1996).

APPENDIX A

Testing Methodologies and Details

A.1 DPSS Throughput Test Methodology

DPSS throughput was measured using a special test client called *tv_sim*. *tv_sim* requests and reads blocks from a DPSS at a user-configurable rate. Since it ignores the data after reading it from the network, *tv_sim* is used to measure the maximum possible transfer rate that a DPSS and client host can achieve.

tv_sim consists of three processes. The *sender* creates and sends block requests to the DPSS master; the requests are randomly chosen from all available blocks in the data set. The *receiver* reads blocks from a single DPSS server; *tv_sim* spawns a receiver for each DPSS server on which a data set is loaded. The parent, or *master*, creates the sender and receivers and controls their shutdown.

The sender's request rate can be controlled at two levels. Requests are broken down into *request lists*. Receipt of a new request list causes the DPSS to flush any outstanding requests from the prior list, as described in Section II.2.3. A user can control both the number of requests per request list and the rate, in lists per second, at which request lists are sent to the DPSS.

Each receiver reports its rate of data receipt in Mb/s after every 40 blocks and

in summary form at program completion. (`tv_sim` can run indefinitely—i.e., until the user interrupts the processes—or can be configured to run until some condition has been met, typically a timeout.) On completion, the sender also reports the total number of request lists and block requests sent. A user can thus compare the amount of data requested with the amount received to determine roughly how much flushing (of unfulfilled requests) took place in the DPSS.

For the tests described in Chapter V, `tv_sim` ran on a Sun Microsystems Ultra Enterprise workstation with an OC-12 ATM interface theoretically capable of absorbing 620 Mb/s, the equivalent of four OC-3 interfaces. Even allowing for the same reduction in performance predicted by Cavanaugh [4] for OC-3 (155 Mb/s) interfaces, which suggests that the OC-12 interface's maximum real capacity is closer to 539-540 Mb/s, we can assume that the client's interface was able to absorb the full output of the three servers described in Chapter V, each of which only possessed an OC-3 interface.

Tests were run to determine the maximum request rate that the DPSS described in Chapter V could support without significant flushing. Note that `tv_sim` is designed to operate on TerraVision-style data sets, whose block size never exceeds 48 KB (49,152 bytes), so the block request rate mentioned below is undoubtedly somewhat higher than a client using 64 KB blocks could achieve.

High request list rates resulted in reduced performance because request flushing occurred too frequently, so eventually the request list rate was limited to

between one and five lists per second. The receivers showed maximum absorption—79-80 Mb/s each—at a request rate of four lists per second, with 175 requests per list. Increasing the number of requests per list or lists per second beyond this threshold only increased flushing; there was no increase in data received.

tv_sim is described briefly in [22]; a user guide is available at [18].

A.2 DMRP Test Methodology

To test the number of RTP sessions the recorder and player could support, three audio/video conferences were individually recorded to the DPSS. Each conference was approximately six minutes long and had an average aggregate data rate of 700 Kb/s. Then the player's conference configuration file was modified to replay all three conferences simultaneously to a single audio and single video session so that the three original conferences appeared to be individual contributors to the new conference, and a recorder captured the new conference in its entirety.

The new, three-contributor conference (representing six RTP sessions, three audio and three video) was then replayed to first one, then two, three, and finally four separate conferences, each consisting of a single audio and a single video session, and the recorder was used to capture the four conferences. Because each of the latter conferences consisted of three audio and three video sources,

the four conferences represented twelve audio and twelve video sessions, all running simultaneously and in parallel. While for the DPSS it could not have represented a great burden—the total throughput of the twelve playback conferences, each averaging 700 Kb/s, likely did not exceed 10 Mb/s, so the final combined data transfer rate for the simultaneous playback and recording was almost certainly less than 20 Mb/s—it apparently represented a maximum capacity for some aspect of the player, for attempts to play back more sessions failed because of insufficient operating system resources. The resources in question may have been kernel memory buffers (mbufs), but this is uncertain.

APPENDIX B

DMRP Source Code

This section presents the DMRP's source code. The DMRP consists of 41 files comprising 12,157 lines of C++ code, according to the Unix utility *wc*; approximately 10,000 of these lines implement the 18 classes that perform most of the DMRP's work, with the bulk of the remainder being code specific either to the recorder or to the player. (A small amount of code represents functionality shared by both that is not part of a class.) No measurement has been made to determine how many of the lines of "code" are comments.

For brevity, supporting or contextual material (exception handler code, preprocessor directives such as *#includes* of system and application header files and *#ifdef* statements, debugging code, and some comments) have been omitted.

B.1 DMRP Shared Classes

Only the class interfaces are presented here.

B.1.1 Networking Classes

B.1.1.1 NetAddr

```
class NetAddr
{
public:
    virtual char *get_hostname() = 0;
```

```
};
```

B.1.1.2 IPAddr

```
class IPAddr : public NetAddr
{
private:
    struct sockaddr_in  addr;
    char                *net2dotted(struct in_addr &);

public:
    IPAddr(struct sockaddr_in &sin)
    {
        memcpy(&addr, &sin, (int)sizeof(addr));
    }
    IPAddr(struct in_addr &a, u_short p);
    IPAddr(const char *host, u_short p);
    IPAddr(const IPAddr &);
    IPAddr()
    {
        memset(&addr, 0, (int)sizeof(addr));
        addr.sin_family = AF_INET;
    }
    IPAddr & operator=(const IPAddr &);
    const struct sockaddr_in *get_addr() { return &addr; }
    u_short get_port() { return ntohs(addr.sin_port); }
    u_short get_port_nbo() { return addr.sin_port; }
    char *get_hostname();
    char *get_ipstring() { return(net2dotted(addr.sin_addr)); }
    void set_addr(const struct sockaddr_in &sin)
    {
        memcpy(&addr, &sin, (int)sizeof(addr));
    }
    void set_hostaddr(const struct in_addr &hostaddr)
    {
        memcpy(&addr.sin_addr, &hostaddr,
            (int)sizeof(struct in_addr));
    }
    void set_port(u_short newport)
    {
        addr.sin_port = htons(newport);
    }
    void set_port_nbo(u_short newport)
    {
        addr.sin_port = newport;
    }
    friend int
```

```

        operator==(const IPAddr &first, const IPAddr &second)
        {
            return(first.addr.sin_addr.s_addr ==
                   second.addr.sin_addr.s_addr);
        }
    };

```

B.1.1.3 NetAP

```

// Network access point ("NetAP")
class NetAP
{
public:
    virtual int recv(char *buf, int len) = 0;
    virtual int send(const char *buf, int len) = 0;
};

```

B.1.1.4 IPNetAP

```

enum ConnType
{
    CONNLESS, CONN
};

class IPNetAP : public NetAP
{
private:
    IPAddr *local,
           *remote;

    int skt;
    int sskt;
    ConnType conntype;
    int ttl;
    fd_set rdset,
           wrset,
           excset;
    int nonblock(int fd);

public:
    IPNetAP(IPAddr &, IPAddr &, ConnType, int);
    IPNetAP();
    ~IPNetAP();
    inline int get_skt() { return(skt); }
    inline int get_sskt() { return(sskt); }
    inline int is_connected()
    {
        if (skt < 0)

```

```

        return(0);
    else
        return(1);
}
void set_localaddr(IPAddr &l);
void set_remoteaddr(IPAddr &r);
void set_conntype(ConnType ct) { conntype = ct; }
void set_ttl(int new_ttl) { ttl = new_ttl; }
void simple_connect();
void server_setup();
int simple_accept();
int mcast_connect(IPAddr &r);
int mcast_connect(void);
void disconnect() { (void)close(skt); skt = sskt; }
/*
 * Polling calls are a little odd. "poll_for_what()" takes
 * "skt" and adds it to the fd_sets specified by "mask."
 * "poll()" only looks at the fd_sets specified by "which,"
 * and returns 0 on a timeout (specified by "to").
 */
int poll_for_what(int mask);
int poll(int which, struct timeval *to);
int recv(char *buf, int len);
int recvfrom(char *buf, int len, IPAddr *fromhost);
int send(const char *buf, int len);
int sendto(const char *buf, int len, IPAddr &tohost);
const IPAddr *localip() { return local; }
const IPAddr *remoteip() { return remote; }
};

```

B.1.2 RTP/RTCP Classes

B.1.2.1 RTPSource

```

/*
 * XXX A bastardized mix of RFC 1889 sample data
 * structures and my own additions. At some point, this
 * should be cleaned up, perhaps eliminating all traces
 * of the sample code.
 */
class RTPSource
{
    friend class RTPDataHandler;
    friend class RTPCtrlHandler;

private:
    rtp_source srcinfo;

```

```

ui32 ssrc;
IPAddr *srcaddr;
char fmt;           // types defined in RTPCommon.h
char pt;
// XXX Do next two duplicate rtp_source info?
ui8  rcvd_data;
ui8  sent_data;
ui32 npkts_sent;
ui32 nocts_sent;
ui32 ntp_sec0;
ui32 ntp_frac0;
ui32 last_ntp_sec;
ui32 last_ntp_frac;
ui32 rtp_ts0;
ui32 rtp_ts;
ui32 ticks_per_us;
ui32 last_rtp_ts;
ui32 first_saved_rtp_ts;
ui16 last_seq;

private:
    inline void set_last_rtp_ts(ui32 t) { last_rtp_ts = t; }

protected:
    void init_seq(ui16);
    inline ui8  get_pt(void)           { return(pt); }
    inline void set_pt(ui8 new_pt)     { pt = new_pt; }
    inline ui8  get_rcvd_data(void)    { return(rcvd_data); }
    inline void set_rcvd_data(ui8 v=1) { rcvd_data = v; }
    inline void set_rtp_ts(ui32 ts)    { rtp_ts = ts; }
    inline void max_seq(ui16 n)        { srcinfo.max_seq = n; }
    inline void cycles(ui32 n)        { srcinfo.cycles = n; }
    inline void base_seq(ui32 n)       { srcinfo.base_seq = n; }
    inline void bad_seq(ui32 n)        { srcinfo.bad_seq = n; }
    inline void received(ui32 n)       { srcinfo.received = n; }
    inline void expected_prior(ui32 n)
    {
        srcinfo.expected_prior = n;
    }
    inline void received_prior(ui32 n)
    {
        srcinfo.received_prior = n;
    }
    inline void transit(ui32 n)        { srcinfo.transit = n; }
    inline void jitter(ui32 n)         { srcinfo.jitter = n; }
    inline rtp_source *get_srcinfo()   { return(&srcinfo); }
    inline void set_ssrc(ui32 n)       { ssrc = n; }
    int set_srcaddr(const IPAddr &newaddr);

```

```

inline void set_sent_data(ui8 yn)    { sent_data = yn; }
inline void set_npkts_sent(ui32 n)   { npkts_sent = n; }
inline void add_npkts_sent(ui32 n)   { npkts_sent += n; }
inline void set_nocts_sent(ui32 n)   { nocts_sent = n; }
inline void add_nocts_sent(ui32 n)   { nocts_sent += n; }
inline void set_ticks_per_us(ui32 n) { ticks_per_us = n; }
inline void set_last_ntp_sec(ui32 n) { last_ntp_sec = n; }
inline void set_last_ntp_frac(ui32 n) { last_ntp_frac = n; }
inline void set_last_seq(ui16 n)     { last_seq = n; }
inline void set_first_saved_rtp_ts(ui32 n)
{
    first_saved_rtp_ts = n;
}

public:
    inline RTPSource(ui32 new_ssrc = 0, ui16 seq = 0)
    {
        RTPSource(NULL, new_ssrc, seq);
    }
    RTPSource(const IPAddr *, ui32, ui16);
    inline ui16 max_seq() { return(srcinfo.max_seq); }
    inline ui32 cycles() { return(srcinfo.cycles); }
    inline ui32 base_seq() { return(srcinfo.base_seq); }
    inline ui32 bad_seq() { return(srcinfo.bad_seq); }
    inline ui32 probation() { return(srcinfo.probation); }
    inline ui32 received() { return(srcinfo.received); }
    inline ui32 expected_prior() { return(srcinfo.expected_prior); }
    inline ui32 received_prior() { return(srcinfo.received_prior); }
    inline ui32 transit() { return(srcinfo.transit); }
    inline ui32 jitter() { return(srcinfo.jitter); }
    inline ui32 get_ssrc() { return(ssrc); }
    inline const IPAddr *get_addr() { return(srcaddr); }
    inline ui8 get_sent_data() { return(sent_data); }
    inline ui32 get_npkts_sent() { return(npkts_sent); }
    inline ui32 get_nocts_sent() { return(nocts_sent); }
    inline ui32 get_ticks_per_us() { return(ticks_per_us); }
    inline ui32 get_last_ntp_sec() { return(last_ntp_sec); }
    inline ui32 get_last_ntp_frac() { return(last_ntp_frac); }
    inline ui32 get_ntp_sec0() { return(ntp_sec0); }
    inline ui32 get_ntp_frac0() { return(ntp_frac0); }
    inline ui32 get_rtp_ts0() { return(rtp_ts0); }
    inline ui32 get_rtp_ts(void) { return(rtp_ts); }
    inline ui32 get_last_rtp_ts(void) { return(last_rtp_ts); }
    inline ui32 get_first_saved_rtp_ts(void)
    {
        return(first_saved_rtp_ts);
    }
    inline ui16 get_last_seq() { return(last_seq); }

```

```

        void print_state();
};

```

B.1.2.2 RTPSourceList

```

// map<key-type, value-type, comparator<key-type> >
typedef map<ui32, RTPSource, less<ui32> > rtpsrcmap;

class RTPSourceList
{
private:
    ui32 dbg;
    rtpsrcmap list;
    map <int, ui32, less<int> > ssrccids;

protected:
    void add(ui32, const IPAddr *);
    int check(ui32);

public:
    RTPSourceList(ui32 level=1)      { dbg = level; }
    inline void  debug(ui32 level=1) { dbg = level; }
    inline ui32  nsrccs()             { return(list.size()); }
    RTPSource *get_source(ui32, const IPAddr *);
    ui32 get_ssrc(ui32 i);
    inline void print_stats(RTPSource *s) { s->print_state(); }
    void print_stats(void);
};

```

B.1.2.3 RTPSession

```

class RTPSession
{
    // Variables
private:
    ui32 dbg;
    ui32 rand_seed;      // to use as arg to random32() later
    RTPSourceList members;
    ui32 bw;             // bandwidth for session, bytes/s

    // Methods
public:
    RTPSession(ui32 level=1);
    inline void debug(ui32 level=1) { dbg = level; }
    inline void set_bw(ui32 bandwidth) { bw = bandwidth; }
    inline ui32 get_bw(void) { return bw; }
    inline RTPSource *get_member(ui32 ssrc, const IPAddr *addr)

```

```

{
    return(members.get_source(ssrc, addr));
}
inline ui32 get_member_ssrc(ui32 i)
{
    return(members.get_ssrc(i));
}
inline ui32 get_n_members(void) { return(members.nsrcs()); }
inline const RTPSourceList &get_members(void)
{
    return(members);
}
inline void print_stats(RTPSource *s) { s->print_state(); }
inline void print_stats(void) { members.print_stats(); }
inline int ssrc_in_use(ui32 ssrc)
{
    ui32 i;

    for (i = 0; i < get_n_members(); ++i)
        if (ssrc == get_member_ssrc(i)) return(1);
    return(0);
}
};

```

B.1.2.4 RTPFormat

```

#define NTYPES 256

class RTPFormat
{
    // variables
private:
    typedef struct
    {
        char *name;
        int sample_rate; // units: Hz or frames/s
        int bps; // bits/sample; -1 if not sample-based
        int bw; // bandwidth
        int reserved1; // not used
        int reserved2; // not used
    }
    _format;

    /*
     * Each static payload in RFC 1890 has a number assigned
     * to it; that number is used as the index into f[]
     */
}

```



```

    _format f[NTYPES];

    // routines
private:
    void init_type(int, char *, int, int, int);

public:
    RTPFormat(void);
    int  is_defined(int);      // is format defined?
    int  get_bps(int);         // get bits/s
    int  get_srate(int);       // get sample rate (Hz)
    int  get_bw(int);          // get bandwidth
    int  ticks2ms(ui8, ui32);
    int  ms2ticks(ui8, ui32);
};

```

B.1.2.5 Timer

```

class Timer
{
protected:
    int send_now;
    timeval ses_start;
    timeval last_pkt;
    timeval next_pkt;
    timeval time_to_next_pkt;

public:
    Timer()
    {
        send_now = 0;
        memset(&last_pkt, 0, sizeof(timeval));
        memset(&next_pkt, 0, sizeof(timeval));
        memset(&time_to_next_pkt, 0, sizeof(timeval));
        (void)gettimeofday(&ses_start, 0);
    }
    inline void set_start_time(timeval start)
    {
        ses_start.tv_sec = start.tv_sec,
        ses_start.tv_usec = start.tv_usec;
    }
    inline const timeval &send_time(void) { return(next_pkt); }
    inline const timeval &last_send(void) { return(last_pkt); }
    inline int ready_now(void) { return(send_now); }
    inline int ready_to_send(void)
    {
        if (send_now)

```

```

    {
        return(1);
    }
    timeval tv;
    (void)gettimeofday(&tv, NULL);
    int late = ((tv.tv_sec > next_pkt.tv_sec)
                || ((tv.tv_sec == next_pkt.tv_sec)
                    && (tv.tv_usec >= next_pkt.tv_usec)));
    return(late);
}
inline timeval *time_to_next_send(void)
{
    (void)gettimeofday(&time_to_next_pkt, NULL);
    if (time_to_next_pkt.tv_sec > next_pkt.tv_sec)
    {
        memset(&time_to_next_pkt, 0, sizeof(timeval));
    }
    else
    {
        time_to_next_pkt.tv_sec =
            next_pkt.tv_sec - time_to_next_pkt.tv_sec;
        int diff = next_pkt.tv_usec - time_to_next_pkt.tv_usec;
        if (diff < 0)
        {
            if (time_to_next_pkt.tv_sec)
            {
                --time_to_next_pkt.tv_sec;
                time_to_next_pkt.tv_usec = diff + 1000000;
            }
            else
            {
                memset(&time_to_next_pkt, 0, sizeof(timeval));
            }
        }
        else
        {
            time_to_next_pkt.tv_usec = diff;
        }
    }
    return(&time_to_next_pkt);
}
inline void set_send_timer(ui32 ms)
{
    if (!ms)
    {
        send_now = 1;
        return;
    }
}

```

```

        last_pkt.tv_sec =
            next_pkt.tv_sec, last_pkt.tv_usec = next_pkt.tv_usec;
// Assumes "ms" is interval between session start and
// current packet
// Should I worry about overflow in the multiplication
// or addition?
int offset = ses_start.tv_usec+ms*1000;
next_pkt.tv_sec = ses_start.tv_sec + offset/1000000;
next_pkt.tv_usec = offset % 1000000;
return;
}
inline void send_wait(void)
{
    timeval tv;
    (void)gettimeofday(&tv, NULL);
    int diff = (next_pkt.tv_sec - tv.tv_sec) * 1000000
        + (next_pkt.tv_usec - tv.tv_usec);
    if (diff > 0)
    {
        tv.tv_sec = diff / 1000000;
        tv.tv_usec = diff % 1000000;
        (void)select(0, NULL, NULL, NULL, &tv);
    }
    return;
}
};

```

B.1.2.6 RTPHandler

```

typedef map <ui32, IPNetAP *, less<ui32> >ipnetapmap;

class RTPHandler
{
    // variables
private:
    IPAddr mcast_addr;
    ui32 my_ssrc;
    ipnetapmap list;
    map <int, ui32, less<int> > ssrcids;
    int ttl;
    IPNetAP *my_ap;
    int dbg;

protected:
    IPAddr lastpkt_src;    // transport addr of last packet's source

protected:

```

```

IPNetAP *create_new_ap(void);
int rm_ap(ui32);
int add_ap(ui32);
int add_ap(ui32, IPNetAP *);
int check_ap(ui32);

public:
    RTPHandler(IPAddr *);
    inline void set_debug(int d) { dbg = d; }
    void set_ttl(int new_ttl) { ttl = new_ttl; }
    inline int set_mcast_addr(IPAddr *rem_addr)
    {
        mcast_addr = *rem_addr;
        return(0);
    }
    inline ui32 naps() { return(list.size()); }
    inline IPAddr get_pktsrc() { return(lastpkt_src); }
    IPNetAP *get_ap(ui32);
    ui32 get_ssrc(ui32);
    int is_a_source(ui32);
    int is_connected(ui32);
    int connect_to_session(ui32);
    int connect_myself(ui32);
    int disconnect_from_session(ui32);
    int disconnect_from_session(void);
    int get_my_skt(void);
    int pktread(char *, int, IPAddr *);
    int replace_ssrc(ui32, ui32);
};

```

B.1.2.7 RTPDataHandler

```

class RTPDataHandler: public RTPHandler, public Timer,
                     public RTPFormat
{
private:
    ui32  dbg;
    timeval wc;
    timeval first;
    ntp64 rcvd;
    ui32 n_diffs;
    ui32 last_ssrc;
    ui32 last_offset;

private:
    /*
     * NB: because this is called from update_seq() and is

```

```

    * *not* used as a real "initialization" method, don't
    * initialize probation to MIN_SEQUENTIAL or you'll never
    * get a correct base_seq.
    */
inline void init_seq(RTPSource *s, ui16 seq)
{
    s->init_seq(seq);
}
int update_seq(RTPSource *, ui16);
int update_jitter(RTPSource *, ui32, ui8);

public:
RTPDataHandler(int req_ttl): RTPHandler(NULL), dbg(0)
{
    reset();
    set_ttl(req_ttl);
}
inline void reset(void)
{
    wc.tv_sec = wc.tv_usec = 0;
    first.tv_sec = first.tv_usec = 0;
    rcvd.upper = rcvd.lower = 0;
    n_diffs = 0;
    IPAddr blank;           // zero-fills a new IPAddr
    lastpkt_src = blank;    // obliterates old IPAddr
    last_ssrc = 0;
    last_offset = 0;
}
inline void debug(ui32 level=1) { dbg = level; }
inline ntp64 &get_rcvd() { return(rcvd); }
inline int pktread(char *buf, int bufsz)
{
    int ret = RTPHandler::pktread(buf, bufsz, &lastpkt_src);
    rcvd = ntp64time();
    return(ret);
}
int pktsend(char *, int, RTPSession *);
RTPSource *get_pkt_source(char *, RTPSession *);
int parse_pkt(char *, int, RTPSession *);
int make_pkt(char *, RTPSession *);
};

```

B.1.2.8 RTPCtrlHandler

```

// RTCP common header word
typedef struct rtcp_common_t
{

```

```

    unsigned int version:2;    /* protocol version */
    unsigned int p:1;          /* padding flag */
    unsigned int count:5;      /* varies by packet type */
    unsigned int pt:8;         /* RTCP packet type */
    ui16 length;               /* pkt len in words, w/o this word */
};

// Reception report block
typedef struct rtcp_rr_t
{
    ui32 ssrc;                 /* data source being reported */
    unsigned int fraction:8;    /* fraction lost since last SR/RR */
    int lost:24;               /* cumul. no. pkts lost (signed!) */
    ui32 last_seq;             /* extended last seq. no. received */
    ui32 jitter;               /* interarrival jitter */
    ui32 lsr;                  /* last SR packet from this source */
    ui32 dlsr;                 /* delay since last SR packet */
};

// SDES item
typedef struct rtcp_sdes_item_t
{
    ui8 type;                  /* type of item (rtcp_sdes_type_t) */
    ui8 length;                /* length of item (in octets) */
    char data[1];              /* text, not null-terminated */
};

/* sender report (SR) */
struct rtcp_sr
{
    ui32 ssrc;                 /* sender generating this report */
    ui32 ntp_sec;              /* NTP timestamp */
    ui32 ntp_frac;
    ui32 rtp_ts;              /* RTP timestamp */
    ui32 psent;                /* packets sent */
    ui32 osent;                /* octets sent */
    rtcp_rr_t rr[1];          /* variable-length list */
};

/* reception report (RR) */
struct rtcp_rr
{
    ui32 ssrc;                 /* receiver generating this report */
    rtcp_rr_t rr[1];          /* variable-length list */
};

/* source description (SDES) */
struct rtcp_sdes

```

```

{
    ui32 src;          /* first SSRC/CSRC */
    rtcp_sdes_item_t item[1]; /* list of SDES items */
};

/* BYE */
struct rtcp_bye
{
    ui32 src[1];      /* list of sources */
};

// One RTCP packet
struct rtcp_t
{
    rtcp_common_t common;
    union rtcp_rpt_type
    {
        struct rtcp_sr sr;
        struct rtcp_rr rr;
        struct rtcp_sdes sdes;
        struct rtcp_bye bye;
    }
    r;
};

typedef struct
{
    char *cname;
    char *name;
    char *email;
    char *phone;
    char *loc;
    char *tool;
    char *note;
}
sdesinfo;

/*
 * The player, for efficiency, should have as much control
 * info as possible available if a source suddenly needs to
 * send data. Ergo, as we parse the saved packet stream, we
 * should build a table--a map, in this case--to correspond
 * each source's SSRC ID with its original CNAME and NAME,
 * modified to reflect the actual source (the player).
 */
typedef map <ui32, sdesinfo, less<ui32> >sdesinfomap;

const int MRP_RTCP_MAXPKTSIZE = 1024;      // 1024 is a guess

```

```

class RTPCtrlHandler: public RTPHandler, public Timer,
                      public RTPFormat
{
    friend class RTPComm;

private:
    int  errcode;
    ui32 dbg;
    int  avg_rtcp_size;
    sdesinfo si;
    sdesinfomap sdes_map;
    ui16 sdes_item_flag;
    timeval last_sr_wc;
    float last_ticks_per_ms;
    ui32 n_ticks;
    ui32 n_srs;
    ui32 avg_ticks_per_rpt;
    char packet[MRP_RTCP_MAXPKTSIZE];

private:
    void rtcp_sr_ntoh(rtcp_sr *);
    void rtcp_sr_hton(rtcp_sr *p);
    void rtcp_rr_t_ntoh(rtcp_rr_t *);
    void rtcp_rr_t_hton(rtcp_rr_t *);
    void rtcp_rr_ntoh(rtcp_rr *, int);
    void rtcp_rr_hton(rtcp_rr *, int);
    void rtcp_sdes_ntoh(rtcp_sdes *);
    void rtcp_sdes_hton(rtcp_sdes *);
    void rtcp_bye_ntoh(rtcp_bye *, int);
    void rtcp_bye_hton(rtcp_bye *, int);
    int validate_hdr(rtcp_t *, ui32);
    char *make_userhost_str(char *);
    int fill_sdesinfo(const char *, const char *);
    int parse_defaults_entry(char *);
    int change_cname(char *);
    int change_name(char *);
    int passwd2sdesinfo(passwd *, const char *, const char *);
    void set_rtcp_interval(int members, int senders,
                           double rtcp_bw, int we_sent,
                           int packet_size, int initial=0);

    // Packet reading routines
    int parse_sr(rtcp_t *, RTPSession *, const IPAddr *);
    int parse_rr(rtcp_t *, RTPSession *, const IPAddr *);
    int parse_sdes(rtcp_t *, RTPSession *, const IPAddr *);
    int parse_bye(rtcp_t *, RTPSession *);
    // Packet creation routines
    inline ui32 calc_dlsr(void);

```



```

void make_sr(RTPSource *src, rtcp_t *);
void make_rr_int(RTPSource *, rtcp_rr_t *);
void make_fixed_rr_hdr(RTPSource *, rtcp_t *);
void choose_sdes_items(rtcp_sdes_type_t[], char *[], int[],
                      int *, sdesinfo *);
int make_sdes_int(char *, ui32, int, rtcp_sdes_type_t[],
                  char *[], int[]);
int make_sdes(char *, int, ui32);
int make_bye(RTPSession *, ui32);

public:
    RTPCtrlHandler(char *, int);
    inline void debug(ui32 level=1) { dbg = level; }
    void print_stats(RTPSource *s);
    void print_stats(RTPSession *ses);
    void errmsg(char *);
    inline ui32 nsrcs() { return(sdes_map.size()); }
    void update_rtp2ntp(rtcp_sr *, RTPSource *);
    int parse_pkt(char *, ui32, RTPSession *);
    int send_rpt(RTPSession *, RTPSource *);
    int parse_saved_pkt(char *, ui32, RTPSession *);
    int add_sdes(ui32);
    sdesinfo *get_sdes(ui32);
    int update_sdes(rtcp_t *, int, RTPSource *);
    inline int pktread(char *buf, int bufsz)
    {
        return(RTPHandler::pktread(buf, bufsz, &lastpkt_src));
    }
    int send_bye(RTPSession *);
    int send_bye(RTPSession *, ui32);
};

```

B.1.2.9 RTPComm

```

class RTPComm: public RTPSession
{
    // variables
private:
    int commdebug;
    int status;      // connected or not connected
    RTPSource *me;
    RTPDataHandler *dh;
    RTPCtrlHandler *ch;
    ui32 good_datapkts;
    ui32 bad_datapkts;
    ui32 good_ctrlpkts;
    ui32 bad_ctrlpkts;

```

```

timeval ses_start;
int  rtcp_send_index;  /* for player; the current
                        player-source that should send
                        the RTCP packet */

// methods
protected:
    int mconn_establish(RTPHandler *, IPAddr *);
    void stamp_pkt(char *, int, RTPPktType);

public:
    inline void set_debug(int d) { commdebug = d; }
    inline void set_full_debug(int d)
    {
        commdebug = d;
        RTPSession::debug(d);
        dh->debug(d);
    }
    RTPComm(char *, int req_ttl=DMRP_DEFAULT_TTL, ui32 req_bw=0);
    inline ~RTPComm(void)
    {
        closecomm();
    }
    inline ui32 ngood(void) { return(good_datapkts); }
    inline ui32 nbad(void) { return(bad_datapkts); }
    inline ui32 ngoodctrl(void) { return(good_ctrlpkts); }
    inline ui32 nbadctrl(void) { return(bad_ctrlpkts); }
    inline int set_name(char *name)
    {
        return(ch->change_name(name));
    }
    inline int set_cname(char *cname)
    {
        return(ch->change_cname(cname));
    }
    inline void set_start_time(void)
    {
        (void)gettimeofday(&ses_start, 0);
    }
    inline void set_start_time(timeval tv)
    {
        memcpy(&ses_start, &tv, sizeof(timeval));
        ch->set_start_time(tv);
        dh->set_start_time(tv);
    }
    inline timeval get_start_time(void) { return(ses_start); }
    inline int get_data_skt(void) { return(dh->get_my_skt()); }
    inline int get_ctrl_skt(void) { return(ch->get_my_skt()); }

```

```

void closecomm(void);
void create_my_source(int);
int join(IPAddr *, IPAddr*);
int join(IPAddr *);
int join(char *addr, ui16 port);
int register_addr(char *, ui16);
int read_data(char *, ui32);
int read_ctrl(char *, ui32);
void wait_until(ui32 offset);
inline int ctrl_ready(void) { return(ch->ready_to_send()); }
inline int data_ready(void) { return(dh->ready_to_send()); }
// XXX This is OK for the listener, not for the player
void send_ctrl(int which=-1);
inline timeval *time_to_next_ctrl(void)
{
    return(ch->time_to_next_send());
}
inline int pkt_type(char *buf)
{
    PktInfo *pi_p = (PktInfo *)buf;
    return(pi_p->type);
}
inline ui32 pkt_size(char *buf)
{
    PktInfo *pi_p = (PktInfo *)buf;
    return(pi_p->size);
}
inline char *skip_pkt(char *buf)
{
    PktInfo *pi_p = (PktInfo *)buf;
    // Make sure to keep the buffer pointer 32-bit aligned
    return(buf+DMRP_PKTLENSZ+ceil32(pi_p->size));
}
void print_pkt(char *buf);
int make_pkt(char *buf);
inline int send_data_pkt(char *buf, int bufsz)
{
    return(dh->pktsend(buf, bufsz, this));
}
int send_next(PktInfo *, char *);
void wait_send_next(void);
inline void reset(void)
{
    dh->reset();
    set_start_time();
    // XXX Need a way to flush the RTPSession/RTPSourceList
}
};

```

B.1.3 DPSS I/O Classes

B.1.3.1 DPSSState

```
class DPSSState
{
    // variables
public:
    int set_id;
    int nsrvrs;
    int connected;                // don't reconnect if != 0
    IssHandle *ih_p;
    dsmSession ds_p;
    dsmSetInfo dsi_p;             // For request_blocks()
    RequestList *rl_p;
    IssInfo *ii_p;
    BlockMap *bm_p;

    // methods
public:
    DPSSState();
    ~DPSSState();
};
```

B.1.3.2 StorageIOBase

```
class StorageIOBase
{
public:
    virtual int open(char *) = 0;
    virtual int close(void) = 0;
    virtual int read(char *, ui32) = 0;
    virtual int write(char *, ui32) = 0;
    virtual int seek(int, int) = 0;    // seek(offset, whence)
};
```

B.1.3.3 DPSSPOC

```
/*
 * DPSS logical block name.
 *
 * Note that only the "x" field is used now. Using all 128 bytes
 * efficiently doesn't seem to be terribly important right now.
 */
typedef lu_key DPSS_lbn;
```

```

class DPSSPOC: public StorageIOBase
{
    // Variables
private:
    int debug;
    // write-specific vars
    ui32 nblocks_wr;           // number of blocks written
    DPSS_lbn cur_block_wr;     // block being written; DPSSCommon.h
    ui32 alloc_bytes;
    ui32 alloc_blocks;
    // read-specific vars
    ui32 file_sz_bl;          // number of blocks in a loaded set
    ui32 block_sz;            // block size, as stored on DSM
    DPSS_lbn cur_block_rd;     // block being read; DPSSCommon.h
    // general vars
    DPSSState *ds_set;
    char *dpss_host;
    char *dsm_host;
    char *storage_serve;
    int *srvrlist;

    // Methods
private:
    char **buildStringList(char *);
    int getDsmSetInfo(int set_id, dsmSetInfo *);
    int send_block(char *, ui32);
    int clean_up_load(void);

public:
    DPSSPOC(char *dsm=NULL, char *dpss=NULL, int dbg=0);
    ~DPSSPOC();
    inline void set_debug(int d) { debug = d; }
    inline int get_debug(void) { return(debug); }
    int open(char *);
    int close(void);
    int open_dpss(char *host=NULL);
    int get_scribe_addrs(char *);
    int register_set(char *, ui32);
    int reserve_space(DPSSState *);
    int prep_new_set(char *, ui32);
    int open_dpss_w(char *, ui32);
    int connect_scribe(int);
    int connect_dpss_scribes(void);
    int open_dpss_r(char *);
    int open_dpss_r(int);
    int close_dpss(void);
    int write(char *data, ui32 nbytes);
    inline int write_dpss(char *data, ui32 nbytes)

```

```

    {
        return(DPSSPOC::send_block(data, nbytes));
    }
    int read(char *, ui32);
    int request_blocks(ui32);
    int receive_blocks(char *, ui32, ui32 *);
    int read_dpss(char *, ui32, ui32 *);
    int seek(int, int);      // seek(offset, whence)
    int seek_dpss(int, int);
    inline int out_of_space(void)
    {
        return(nblocks_wr >= alloc_blocks);
    }
    inline int get_set_id(void) { return(ds_set->set_id); }
    inline int get_num_blocks(void) { return(file_sz_bl); }
};

```

B.1.4 SessionManager

```

// Make sure DMRP_DPSS_BLOCKSZ is always an even mod of 1024...
const int DMRP_DPSS_BLOCKSZ = 65536;      // bytes
const int DMRP_BUFSZ = 2 * DMRP_DPSS_BLOCKSZ;

/*
 * For recording: don't wait for data packets; for playback:
 * show all original conference members, whether or not they
 * send data (playback not yet implemented)
 */
const ui32 SM_ALL_PKTS    = 0x8;

struct DMRPSessionInfo
{
    char *addr;
    ui16 port;
    int ttl;
    int set_id;
    char *filename;
    ui32 filesz;
    ui32 bandwidth;
    /*
     * For recording; currently seconds. 1 day == 86400 s;
     * 7 days == 604800 s; 30 days == 2592000 s, so an int
     * should be enough space for now. An int also allows
     * negative values if needed, unlike a ui32.
     */
    int duration;
    char *imglib_name;

```

```

};

class SessionManager
{
    // variables
private:
    timeval quit_time;      /* time at which program should
                             end; set if DMRPSessionInfo's
                             "duration" field is non-zero */

public:
    int debug;
    int flags;
    AppStatus as;
    RTPComm *comm;
    DPSSPOC *dpss_poc;
    DMRPSessionInfo si;
    char databuf[DMRP_BUFSZ];
    int dloc;
    char *toolname;
    char *dpss_host;
    char *dsm_host;
    char *dpss_servs;
    int nblocks;

    // methods
private:
    int create_comm(void);

public:
    SessionManager(char *tool, DMRPSessionInfo *si_p, int dbg=0);
    int join_session(void);
    inline void leave_session(void) { comm->closecomm(); }
    int open_dpss(char *file, char *dpss, char *dsm, ui32 sz=0,
                  char *servs=NULL);      /* if defaults used,
                                           assume playback */
    int open_dpss(int setid, char *dpss, char *dsm, ui32 sz=0,
                  char *servs=NULL);      /* if defaults used,
                                           assume playback */

    int close_dpss(void);
    void set_bandwidth(int bw) { si.bandwidth = bw; }
    void set_ttl(int req_ttl) { si.ttl = req_ttl; }
    int set_toolname(char *tn)
    {
        if (tn && (toolname = strdup(tn)) == NULL)
        {
            return(-1);
        }
    }

```

```

        return(0);
    }
int set_sktmask(fd_set *sktmask)
{
    FD_ZERO(sktmask);
    FD_SET(comm->get_data_skt(), sktmask);
    FD_SET(comm->get_ctrl_skt(), sktmask);
    return((comm->get_data_skt() < comm->get_ctrl_skt())
        ? comm->get_ctrl_skt()+1
        : comm->get_data_skt()+1);
}
int process_data_pkt(char *, int);
int process_ctrl_pkt(char *, int);
int save_block(int *, int *, int *);
int mcast_receive(void);
// For invocation via pthread_create()
inline void *mcast_receive_thr(void)
{
    return((void *)mcast_receive());
}
int prepare_data_pkt(char *);
int register_session(void);
int mcast_send(void);
// For invocation via pthread_create()
inline void *mcast_send_thr(void)
{
    return((void *)mcast_send());
}
inline int get_nblocks(void) { return(nblocks); }
inline void quit(void) { as.done(); }
/* Recording: don't wait for data packets to arrive
   to begin saving packets */
inline void save_from_start(void)
{
    flags |= SM_ALL_PKTS;
}
/*
 * Playback: send RTCP packets for all original
 * session members, even if they didn't send any data
 */
inline void show_all_members(void)
{
    flags |= SM_ALL_PKTS;
}
inline void reset(void)
{
    comm->reset();
    as.remove(APPSTAT_DONE);    // See AppStatus
}

```



```
    }
};
```

B.1.5 AppStatus

```
/*
 * Possible values for "status" field in AppStatus; could
 * be extended to allow values like APPSTAT_PLAY,
 * APPSTAT_REC, APPSTAT_FF, APPSTAT_REW, etc.
 */
const ui32 APPSTAT_DONE = 0x1;

class AppStatus
{
    // variables
private:
    pthread_mutex_t mtx;
    ui32 status;      // bitmask

    // methods
protected:
    void lock(void);
    void unlock(void);

public:
    AppStatus(int s=0);
    ~AppStatus(void);
    ui32 get(void);
    void set(ui32);      // overwrites "status" with arg
    void add(ui32);      // adds flags to "status"
    void remove(ui32);   // removes flags from "status"
    ui32 doneq(void);    // return 1 if done, 0 otherwise
    void done(void);     // set "done" flag
    void clear(void);    // reset
};
```

B.2 DMRP common functions

The following are routines that both the recorder and player use that lie outside of any of the shared classes. In general, these routines are used for parsing the conference configuration file discussed in Section IV.3 or for managing the mutex locks required for inter-thread synchronization.

```

/*****/
passwd *
mrp_getpwinfo(void)
{
    passwd *pwd = getpwuid(getuid());
    if (pwd == NULL)
    {
        return(NULL);
    }
    void (*oldnh)() = set_new_handler(&DMRP_new_handler);
    passwd *new_pwd = new passwd;
    set_new_handler(oldnh);    // reinstall default new() handler
    memcpy(new_pwd, pwd, sizeof(passwd));
    return(new_pwd);
}
/* end mrp_getpwinfo(void) */

/*****/
// From iss_utils/iss_comm_utils.c
char *
mrp_host2ip(char *hostname)
{
    char *ip;
    struct hostent *hp;

    if ((hp = gethostbyname(hostname)) != NULL)
    {
        if ((ip = inet_ntoa(*(struct in_addr *) (hp->h_addr_list[0])))
!= NULL)
        {
            if ((ip = strdup(ip)) != NULL)
            {
                return(ip);
            }
        }
    }
    return(NULL);
}
/* end mrp_host2ip */

/*****/
char *
strip_token(char *str, char fs)
{
    if (!str)
        return(NULL);

```

```

        char *tmp = strchr(str, fs);
        if (tmp != NULL)
        {
            *tmp = '\\0';
            ++tmp;
        }
        return(tmp);
    }
// end strip_token(char *, char)

/*****/
char *
skip_token(char *str, char fs)
{
    if (!str)
        return(NULL);
    char *tmp = strchr(str, fs);
    if (tmp != NULL)
    {
        ++tmp;
    }
    return(tmp);
}
// end skip_token(char *, char)

/*****/
void
replace_whitespace(char *str, char rch)
{
    if (!str)
        return;
    for (int i = 0; i < strlen(str); ++i)
    {
        if (isspace(str[i]))
            str[i] = rch;
    }
    return;
}
// end replace_whitespace(char *, char)

/*****/
int
get_addr_info(char *str, char **addr, u_short *port)
{
    char *func = "get_addr_info()",

```

```

        *ptr;

        // Assume "str" is of standard form "ADDR/PORT"
        if ((ptr = strchr(str, '/')) == NULL)
        {
            cerr << func << ": invalid multicast address \"" << str <<
"\\"?\\n"
            << flush;
            return(-1);
        }
        ptr[0] = '\\0';
        ++ptr;
        if ((*addr = strdup(str)) == NULL)
        {
            cerr << "strdup() failure\\n" << flush;
            return(-1);
        }
        // XXX Consider a weak check for port using isdigit() on first
character
        *port = atoi(ptr);
        ptr[-1] = '/';
        return(0);
    }
    // end get_addr_info(char *, char **, u_short *)

/*****/
int
read_conference_file(char *conffile, DMRPSessionInfo **sinfo)
{
    char *func = "read_conference_file()";

    *sinfo = NULL;
    FILE *fp = fopen(conffile, "r");
    if (fp == NULL)
    {
        cerr << "Couldn't open " << conffile << endl;
        return(-1);
    }
    char  buffer[CONFLINELEN],
        *tok1,
        *tok2,
        *tok3,
        *bufptr,
        *sesname = NULL,
        *imglib_name = NULL;
    int medianum,
        num_media = 0,

```

```

        lineno,
        duration = 0; /* 0 implies recorder should run
                        forever; no effect on player,
                        at present */

/*
 * The parsing is pretty primitive, and will break
 * if the input is not newline-terminated.
 */
for (lineno = 1; fgets(buffer, CONFLINELEN, fp); ++lineno)
{
    // Strip out the newline.
    char *nl = strrchr(buffer, '\n');
    if (!nl)
    {
        cerr << func << ": line " << lineno << " not
newline-terminated "
        "\n\tor is > " << CONFLINELEN << " characters\n";
        return(-1);
    }
    *nl = '\0';
    if ((bufptr = strip_token(buffer, '=')) == NULL)
    {
        continue; // unrecognized line, discarded
    }
    tok1 = buffer;
    if (!strcmp(tok1, "session_name"))
    {
        replace_whitespace(bufptr, '_'); // underscore
replaces whitespace
        if ((sesname = strdup(bufptr)) == NULL)
        {
            cerr << func << ": strdup() for session name
failed\n";
            (void)fclose(fp);
            return(-1);
        }
    }
    else if (!strcmp(tok1, "session_duration"))
    {
        // session_duration in minutes, but duration in seconds
        duration = atoi(bufptr) * 60;
        if (!duration)
        {
            cerr << func << ": invalid value \"" << bufptr
            << "\" for session duration\n";
            (void)fclose(fp);
            return(-1);
        }
    }
}

```

```

    }
}
else if (!strcmp(tok1, "imglib_name"))
{
    replace_whitespace(bufptr, '_');    // underscore
replaces whitespace
    if ((imglib_name = strdup(bufptr)) == NULL)
    {
        cerr << func << ": strdup() for ImgLib name
failed\n";
        (void)fclose(fp);
        return(-1);
    }
}
else if (isdigit(tok1[0]))    // begin media-specific
handling
{
    // Media numbers start w/ 1, so make sure to adjust
    if ((medianum = (atoi(tok1)-1)) >= num_media)
    {
        DMRPSessionInfo *tmp;
        tmp = (DMRPSessionInfo *)realloc((*sinfo),
            (medianum+1)*sizeof(DMRPSessionInfo));
        if (tmp == NULL)
        {
            cerr << func << ": Couldn't create or extend
DMRPSessionInfo "
            "array\n";
            (void)fclose(fp);
            return(-1);
        }
        *sinfo = tmp;
        for (int index = num_media; index < medianum+1;
++index)
        {
            // initialize to impossible value
            (*sinfo)[index].set_id = -1;
        }
        num_media = medianum+1;
    }
    tok2 = bufptr;
    if ((tok3 = strip_token(bufptr, '=')) == NULL)
    {
        // malformed media line
        cerr << "Skipping malformed media line: \"" << tok1
<< "=\""
        << bufptr << "\"\n";
        continue;
    }
}

```

```

    }
    if (!strcmp(tok2, "media_port"))
    {
        (*sinfo)[medianum].port = atoi(tok3);
    }
    else if (!strcmp(tok2, "media_ip"))
    {
        if (((*sinfo)[medianum].addr = strdup(tok3)) == NULL)
        {
            cerr << func << ": strdup() for multicast address
failed\n";

            (void)fclose(fp);
            return(-1);
        }
    }
}
else if (!strcmp(tok2, "media_name"))
{
    if (sesname)
    {
        (*sinfo)[medianum].filename
            = (char *)calloc(strlen(sesname)
                            +strlen(tok3)
                            +strlen(tok1)+3,
                            sizeof(char));
    }
    else
    {
        (*sinfo)[medianum].filename
            = (char *)calloc(strlen("<UNKNOWN>")
                            +strlen(tok3)
                            +strlen(tok1)+3,
                            sizeof(char));
    }
    if ((*sinfo)[medianum].filename == NULL)
    {
        cerr << func << ": couldn't create media filename
for "

        << tok3 << ", " << tok1 << endl;
        (void)fclose(fp);
        return(-1);
    }
    if (sesname)
    {
        strcpy((*sinfo)[medianum].filename, sesname);
    }
    else
    {
        strcpy((*sinfo)[medianum].filename, "<UNKNOWN>");
    }
}

```

```

        }
        strcat((*sinfo)[medianum].filename, "_");
        strcat((*sinfo)[medianum].filename, tok3);
        strcat((*sinfo)[medianum].filename, "_");
        strcat((*sinfo)[medianum].filename, tok1);
    }
    else if (!strcmp(tok2, "media_setid"))
    {
        (*sinfo)[medianum].set_id = atoi(tok3);
    }
    else if (!strcmp(tok2, "media_type") && strcmp(tok3,
"1"))
    {
        cerr << func << ": can't handle media type " << tok3
<< endl;
        (void)fclose(fp);
        return(-1);      /* XXX  Throw out sinfo[] entry and
continue
                           instead? */
    }
    else if (!strcmp(tok2, "media_ttl"))
    {
        (*sinfo)[medianum].ttl = atoi(tok3);
    }
} // end media-specific handling
} // end for()
// Fill in session params applying to all media
for (int i = 0; i < num_media; ++i)
{
    (*sinfo)[i].duration = duration;
    /*
    * Copy ImgLib name into each DMRPSessionInfo; for now,
    * the name will not be unique per medium, but only per
    * conference (i.e., group of related sessions)
    */
    if (imglib_name)
    {
        (*sinfo)[i].imglib_name = strdup(imglib_name);
        if ((*sinfo)[i].imglib_name == NULL)
        {
            cerr << func << ": Couldn't copy ImgLib name\n" <<
flush;
            (void)fclose(fp);
            return(-1);
        }
    }
}
}
if (!feof(fp))

```



```

    {
        perror("Error reading session config file");
        return(-1);
    }
    if (fclose(fp))
        cerr << func << ": error closing file (but continuing
anyway)\n";
    if (sesname)
        free(sesname);
    return(num_media);
}
// end read_conference_file(char *, DMRPSessionInfo **)

/*****/
void
mrp_lock_mutex(pthread_mutex_t *mtx_p)
{
    char *func = "mrp_lock_mutex(pthread_mutex_t *)";

    int ret = pthread_mutex_lock(mtx_p);
    if (ret)
    {
        cerr << func << ": ";
        throw DMRPExc(DMRPEXC_MTX_LOCK, ret);
    }
    return;
}
// end mrp_lock_mutex(pthread_mutex_t *)

/*****/
void
mrp_unlock_mutex(pthread_mutex_t *mtx_p)
{
    char *func = "mrp_unlock_mutex(pthread_mutex_t *)";

    int ret = pthread_mutex_unlock(mtx_p);
    if (ret)
    {
        cerr << func << ": ";
        throw DMRPExc(DMRPEXC_MTX_UNLOCK, ret);
    }
    return;
}
// end mrp_unlock_mutex(pthread_mutex_t *)

```

B.3 Recorder

The listener consists of two main code files, `listener.cc` and `listener_utils.cc`. `listener.cc` contains the bulk of the recorder's code, including `main()`; `listener_utils.cc` contains "utility" routines that parse command-line arguments and print the program's usage message.

The recorder was originally called the listener, hence the naming conventions for the files, data types, and some variables.

B.3.1 Common definitions

```
const ui32 LSNR_DEFAULT_FILESZ = 1073741824; // bytes (== 1 GB)
/*
 * Possible flag values. For compatibility with
 * SessionManager, keep listener-only flags in high-order
 * byte of flags variable. SM_* defined in SessionManager.h.
 */
const ui16 LSNR_NOSAVE_DBG = SM_NOSAVE_DBG;
const ui16 LSNR_QUIET      = SM_QUIET;
const ui16 LSNR_ALL_PKTS   = SM_ALL_PKTS;
const ui16 LSNR_HIDE_NAME  = 0x200;      // hide caller's RTPName

const
struct ListenerArgs
{
    int  my_id;
    char *dpss_host;
    char *dsm_host;
    char *dpss_serve;
    ui16 debug;
    ui16 flags;
    int  ttl;
    int  bandwidth;
    int  filesize; // use as size of EACH session file
    int  duration; // in seconds, for recording
    char *imglib_prog; /* ImgLib script/program to
                        register set size, ID */
    char *imglib_name; /* ImgLib "name" of this
```

```

                                collection of sessions */
    char *sesfile;
    int nsessions;
    DMRPSessionInfo *sessions;
};

```

B.3.2 listener.cc

```

// Globals, required because of signal-handling semantics:
char *pname;           // required for DPSS
int nsessions = 0;
SessionManager **smgrs;
int lsnr_debug;
// Needed as long as LSNR_NOSAVE_DBG is around
int lsnr_nosave;
uil6 quiet = 0;
uil6 hide_caller_name = 0;      // by default, use caller's RTPName
sigset_t lsnr_sigmask;
pthread_t *thread_ids = NULL,
          timer_tid;

/*****
int
main(int argc, char **argv)
{
    ListenerArgs la;
    TimerArgs ta;
    char imglib_cmd[512],
          setidstr[10],
          nbytesstr[20];

    if (init(argc, argv, &la) < 0)
    {
        cerr << "Fatal initialization error" << endl;
        exit(-1);
    }
    if ((nsessions = join_sessions(smgrs, la.nsessions))
        != la.nsessions)
    {
        // currently should be fatal because it's easier
        cerr << "Couldn't join all sessions--fatal error\n";
        cleanup(-1);
    }
    if (start_session_threads(smgrs, &thread_ids, nsessions) < 0)
    {
        // currently should be fatal because it's easier
        cerr << "Couldn't start threads--fatal error\n";

```

```

        cleanup(-1);
    }
    /*
     * If we requested a session timeout, it applies to every
     * session; ergo, we set up a separate thread which will
     * do nothing but set an alarm which will be caught by
     * the main thread.
     */
    if (la.sessions[0].duration)
    {
        ta.ns = la.sessions[0].duration;
        int r = pthread_create(&timer_tid, NULL, timer,
                               (void *)&ta);

        if (r)
        {
            cerr << "Error creating timer thread\n" << flush;
            cleanup(-1);
        }
    }
    int signo, ret;
    if ((ret = sigwait(&lsnr_sigmask, &signo)) < 0)
    {
        cerr << strerror(ret) << endl << flush;
        exit(-2);
    }
    int nbytes = -1,
        setid = -1;
    if (signo == SIGINT || signo == SIGALRM)        // all is well
    {
        if (signo == SIGALRM)
        {
            cout << "Conference timeout reached\n" << flush;
        }
        I_am_done();
        if (la.imglib_prog)
        {
            // Initialize ImgLib buffer
            memset(imglib_cmd, 0, 256);
            sprintf(imglib_cmd, "%s \"%s\" ", la.imglib_prog,
                    smgrs[0]->si.imglib_name);
        }
        // Wait for child threads
        for (int i=0; i < nsessions; ++i)
        {
            // debugging
            cout << "Joining thread " << i << "...\\n" << flush;
            if (ret = pthread_join(thread_ids[i], NULL))
            {

```

```

        cerr << "pthread_join(): " << strerror(ret)
              << endl << flush;
        exit(-3);
    }
    close_session(smgrs[i]);
    // For ImgLib/WALDO
    if (!lsnr_nosave)
    {
        if (la.imglib_prog)
        {
            setid = smgrs[i]->dpss_poc->get_set_id();
            nbytes = smgrs[i]->dpss_poc->get_num_blocks()
                    * DMRP_DPSS_BLOCKSZ;
            if (!nbytes)
            {
                nbytes = -1;
                setid = -1;
            }
            // print the set ID & # of bytes to a buffer
            memset(setidstr, 0, 10);
            memset(nbytesstr, 0, 20);
            sprintf(setidstr, "%d ", setid);
            sprintf(nbytesstr, "%d ", nbytes);
            strcat(imglib_cmd, setidstr);
            strcat(imglib_cmd, nbytesstr);
        }
    }
    cout << flush;
}
// Execute the ImgLib script if one is provided
if (la.imglib_prog)
{
    /*
     * Note that system(3s) is *unsafe* to call in
     * a multithreaded environment, so it must not
     * be called at a point where multiple threads
     * are active.
     */
    if (system(imglib_cmd) < 0)
    {
        perror("system()");
        cerr << "Failed ImgLib script and args:\n\t"
              << imglib_cmd << endl << flush;
    }
}
exit(0);
}
else

```

```

    {
        cerr << "Received signal " << signo
              << "; abnormal exit...\n" << flush;
        I_am_done();
        for (int i=0; i < nsessions; ++i)
        {
            (void)pthread_join(thread_ids[i], NULL);
        }
        exit(signo);
    } // end if (signo == SIGINT)/else
    return(0);
}
/* end main */

/*****
/*
* v 0.1: single session, single member
* v 0.2: single session, multiple members
* v 0.3: multiple sessions, multiple members
*/
char *
make_toolname(void)
{
    char *basetoolname = "DMRP listener";
    char *vers = ", v. 0.3";

    char *toolname = new char[strlen(basetoolname)+strlen(vers)+1];
    strcpy(toolname, basetoolname);
    strcat(toolname, vers);
    return(toolname);
}
/* end make_toolname(void) */

/*****
int
init(int ac, char **av, ListenerArgs *la_p)
{
    pname = av[0]; // program name
    // increase file descriptor and stack size limits
    rlimit rl;
    memset(&rl, 0, sizeof(rlimit));
    if (getrlimit(RLIMIT_STACK, &rl) < 0)
    {
        perror("getrlimit()");
        exit(-1);
    }
}

```



```

catch (SessionMgrExc e)
{
    e.print();
    return(-1);
}
/*
 * XXX A hack to make sure functionality ripped
 * out of here and dumped into the SessionManager
 * can print or suppress printing in the same way
 */
smgrs[i]->flags = la_p->flags;
// DPSS-related initialization
/* if using DPSS, file size should be a multiple of
   DPSS block size */
if (!la_p->sessions[i].filesz)
{
    // 16,384 DPSS blocks of 64 KB
    la_p->sessions[i].filesz = LSNR_DEFAULT_FILESZ;
}
else // LSNR_BUFSZ assumed to be same as DMRP_BUFSZ
{
    if (la_p->debug)
        cout << "size " << la_p->sessions[i].filesz
              << ", LSNR_BUFSZ " << LSNR_BUFSZ << endl;
    int x = issRoundUp(la_p->sessions[i].filesz, LSNR_BUFSZ);
    if (la_p->debug)
        cout << x << " blocks\n" << (x * LSNR_BUFSZ) << "
bytes\n";

    smgrs[i]->si.filesz = x * LSNR_BUFSZ;
}
if (!(la_p->flags & LSNR_NOSAVE_DBG))
{
    // debugging
    cout << "Opening DPSS...\n" << flush;
    if (smgrs[i]->open_dpss(la_p->sessions[i].filename,
                          la_p->dpss_host, la_p->dsm_host,
                          la_p->sessions[i].filesz,
                          la_p->dpss_serve) < 0)
    {
        cerr << "Error creating DPSSPOC " << i << endl;
        return(-1);
    }
}
} // end for()
return(0);
}
/* end init(int, char **, ListenerArgs *, RTPSession *) */

```



```

/*****/
/*
 * Required because Solaris native compiler will not allow
 * pthread_create() to call SessionManager::mcast_receive_thr()
 * directly for some reason.
 */
void *
run_session(void *targs)
{
    SessionManager *sm_p = (SessionManager *)targs;
    return(sm_p->mcast_receive_thr());
}
// end run_session(void *)

/*****/
int
start_session_threads(SessionManager **smarray,
                      pthread_t **tids_pp, int ns)
{
    char *func = "start_session_threads(SessionManager **, pthread_t
**, int)";
    pthread_attr_t tattr;
    int err;

    if (err = pthread_attr_init(&tattr))
    {
        cerr << func << ": pthread_attr_init() error ("
            << strerror(err) << ")\n" << flush;
        return(-1);
    }
    if (err = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM))
    {
        cerr << func << ": pthread_attr_setscope() error ("
            << strerror(err) << ")\n" << flush;
        return(-1);
    }
    *tids_pp = (pthread_t *)calloc(ns, (int)sizeof(pthread_t));
    if (*tids_pp == NULL)
    {
        cerr << "Error creating thread ID array; out of memory?\n" <<
flush;
        return(-1);
    }
#ifdef SOLARIS
    thr_setconcurrency(ns+1);    // +1 for signal-watching thread
#endif
}

```

```

        for (int i=0; i < ns; ++i)
        {
            int r = pthread_create((*tids_pp)+i, &tattr,
                                   run_session, smarray[i]);
            if (r)
            {
                cerr << "Error creating thread for session " << i << endl
<< flush;
                // continue?
                return(-1);
            }
        }
        return(0);
    }
    // end start_session_threads(SessionManager **, pthread_t **, int)

    /*****
void
#ifdef OS_SOLARIS_2_4
cleanup(int)
#else
cleanup(...)
#endif
{
    for (int i=0; i < nsessions; ++i)
    {
        close_session(smgrs[i]);
    }
    exit(0);
}
/* end cleanup */

    /*****
void
set_sig_handlers(void)
{
    struct sigaction action;

    memset(&action, 0, (int)sizeof(action));
#ifdef SOL2_5_SIGNALS
    action.sa_handler = cleanup;
#else
    action.sa_handler = (void(*)())cleanup;
#endif
    if (sigaction(SIGINT, &action, NULL) == -1)
    {

```

```

        perror("sigaction");
        cerr << "Unable to set SIGINT handler\n";
        exit(-1);
    }
    return;
}
/* end set_sig_handlers(void) */

/*****
int
join_sessions(SessionManager **sessions, int ns)
{
    char *func = "join_sessions()";
    int nsuccessful = 0;

    struct timeval tv;
    (void)gettimeofday(&tv, NULL);

    for (int i = 0; i < ns; ++i)
    {
        // address and port initialized by memcpy() in init()
        if (sessions[i]->join_session() < 0)
        {
            cerr << "Couldn't join session at address "
                 << sessions[i]->si.addr << "/"
                 << sessions[i]->si.port << endl;
            break;
        }
        else
        {
            // Make sure all sessions have same start time
            sessions[i]->comm->set_start_time(tv);
            /*
             * If the program's caller wants to hide
             * his/her name, go along with it
             */
            if (hide_caller_name)
                sessions[i]->comm->set_name("DMRP recorder");
            ++nsuccessful;
        }
    }
    return(nsuccessful);
}
// end join_sessions(SessionManager **, int)

*****/

```

```

/*
 * NB: Solaris man page suggests you can't make a single call
 * to sigaddset() with a mask of the requested signals, but
 * rather must make a separate call for each signal blocked.
 */
void
set_sigs2block(sigset_t *sigs2block)
{
    if (sigemptyset(sigs2block) < 0)
    {
        perror("sigemptyset()");
        cleanup(0);
    }
    if (sigaddset(sigs2block, SIGINT) < 0)
    {
        perror("sigaddset(SIGINT)");
        cleanup(0);
    }
    // Only needed if timeout requested, but can't hurt to block it
    if (sigaddset(sigs2block, SIGALRM) < 0)
    {
        perror("sigaddset(SIGALRM)");
        cleanup(0);
    }
    return;
}
/* end set_sigs2block(sigset_t *) */

/*****/
void
close_session(SessionManager *sm_p)
{
    if (!quiet)
    {
        cout << sm_p->comm->ngood() << " data packets, "
              << sm_p->comm->ngoodctrl() << " RTCP packets, "
              << sm_p->get_nblocks() << " blocks\n" << flush;
    }
    // Flush any pending data
    if (!lsnr_nosave)
    {
        if (!quiet)
        {
            // turn on the "Don't forget to remove set ..." message
            sm_p->dpss_poc->set_debug(1);
        }
        if (sm_p->close_dpss() < 0)

```

```

        {
            cerr << "Error closing DPSS/DSM connections; continuing
cleanup\n"
                << flush;
        }
    }
    sm_p->leave_session();
    return;
}
// end close_session(SessionManager *)

/*****/
void
I_am_done(void)
{
    char *func = "I_am_done(void)";

    for (int i=0; i < nsessions; ++i)
    {
        smgrs[i]->quit();
    }
    return;
}
// end I_am_done(void)

/*****/
void *
timer(void *targs)
{
    TimerArgs *ta_p = (TimerArgs *)targs;
    alarm(ta_p->ns);
    return((void *)1);
}
// end timer(void *)

```

B.3.3 listener_utils.cc

```

extern char *optarg;
extern int optind;
// for iss_utils
extern int Debug;

/*****/
void
init_ListenerArgs(ListenerArgs *la_p)

```

```

{
    memset(la_p, 0, sizeof(ListenerArgs));
    la_p->my_id = -1;      // a value that will never be valid
    la_p->ttml = DMRP_DEFAULT_TTL;    // MRPCCommon.h
    la_p->filesz = LSNR_DEFAULT_FILESZ;
    /*
     * You can specify AT MOST one session on the command
     * line--multiple sessions must be specified in a
     * session config file (and a single session can be,
     * too). NOTE: if you specify a session config file,
     * any session info on the command line is ignored.
     */
    la_p->sessions =
        (DMRPSessionInfo *)calloc(1, sizeof(DMRPSessionInfo));
    if (la_p->sessions == NULL)
    {
        cerr << "Couldn't allocate DMRPSessionInfo; out of
memory?\n";
        exit(1);
    }
    la_p->nsessions = 1;
    for (int i = 0; i < la_p->nsessions; ++i)
    {
        la_p->sessions[i].set_id = -1;
    }
    return;
}
/* end init_ListenerArgs(ListenerArgs *) */

/*****/
void
parse_args(int ac, char **av, ListenerArgs *la_p)
{
    const char *LSNR_ARGS = "F:I:MS:ab:d:f:hi:l:m:nqs:t:";
    const char *opts = LSNR_ARGS;
    char *prog = av[0];
    int c;

    while ((c = getopt(ac, av, opts)) != EOF)
    {
        switch (c)
        {
            case 'F':
                la_p->sesfile = optarg;
                break;
            case 'I':      // ImgLib script
                la_p->imglib_prog = optarg;

```

```

        break;
case 'M':
    la_p->flags |= LSNR_NOSAVE_DBG;
    break;
case 'S':
    la_p->dpss_servs = optarg;
    break;
case 'a':
    la_p->flags |= LSNR_ALL_PKTS;
    break;
case 'b':
    la_p->bandwidth = (ui32)atoi(optarg);
    break;
case 'd':
    la_p->debug = atoi(optarg);
    // for iss_utils lib
    Debug = 1;
    break;
case 'f':
    la_p->sessions[0].filename = optarg;
    break;
case 'h':
    usage(prog);
    break;
case 'i':
    if (optarg == NULL)
        usage(prog);
    la_p->dpss_host = optarg;
    break;
case 'l':
    if (optarg == NULL)
        usage(prog);
    // option is in minutes, but I need seconds
    la_p->duration = atoi(optarg) * 60;
    break;
case 'm':
    if (optarg == NULL)
        usage(prog);
    la_p->dsm_host = optarg;
    break;
case 'n':
    la_p->flags |= LSNR_HIDE_NAME;
    break;
case 'q':
    la_p->flags |= LSNR_QUIET;
    break;
case 's':
    if (optarg == NULL)

```

```

        usage(prog);
        la_p->filesz = (ui32)atoi(optarg);
        break;
    case 't':
        if (optarg == NULL)
            usage(prog);
        la_p->ttn = atoi(optarg);
        break;
    case '?':
    default:
        usage(prog);
        break;
    }
}
if (la_p->sesfile)
{
    free(la_p->sessions);
    la_p->sessions = NULL;
    // debugging
    cout << "Reading session info...\n";
    la_p->nsessions =
        read_conference_file(la_p->sesfile, &la_p->sessions);
    if (la_p->nsessions <= 0)
    {
        cerr << "Error parsing conference configuration file!\n";
        exit(-2);
    }
    for (int i = 0; i < la_p->nsessions; ++i)
    {
        la_p->sessions[i].ttn = la_p->ttn;
        la_p->sessions[i].bandwidth = la_p->bandwidth;
        la_p->sessions[i].filesz = la_p->filesz;
        /* Any timeout on command line overrides conf.
           file value */
        if (la_p->duration)
            la_p->sessions[i].duration = la_p->duration;
        if (la_p->debug)
            print_DMRPSessionInfo(la_p->sessions+i);
    }
}
if (!(la_p->flags & LSNR_NOSAVE_DBG)
    && (la_p->sessions[0].filename == NULL))
{
    cerr << "To save the data, you must specify a filename using
-f"
        << endl;
    usage(prog);
}

```



```

        "Notes:\n"
        "\t--You must use a session file to record multiple
sessions\n"
        "\t--If not using a session file, must use either -f or -M;
also, \n"
        "\t ADDR/PORT must be last on command line\n\n";
        exit(1);
    }
    /* end usage */

```

B.4 Player

The player consists of two main code files, `player.cc` and `player_utils.cc`. `player.cc` contains the bulk of the player's code, including `main()`; `player_utils.cc` contains "utility" routines that parse command-line arguments and print the program's usage message.

B.4.1 Common definitions

```

/*
 * Possible flag values. For compatibility with
 * SessionManager, keep player-only flags in high-order
 * byte of flags variable. SM_* defined in SessionManager.h.
 */
const int Experimental = SM_NOSEND;
const uil6 PLAYER_QUIET = SM_QUIET;

struct PlayerArgs
{
    char *dpss_host;
    char *dsm_host;
    char *sesfile;
    uil6 flags;
    int ttl;           // multicast ttl
    int bandwidth;
    int nrepeats;      // # of times to repeat playback; -1 == forever
    uil6 d;             // _d_ ebug value
    char v;            // _v_ erbose, on/off
    int nsessions;
    DMRPSessionInfo *sessions;
}

```

```
};
```

B.4.2 *player.cc*

```
// Globals, required because of signal-handling semantics:
char *pname;      // required for DPSS
int nsessions;
SessionManager **smgrs;
uil6 quiet      = 0;
sigset_t player_sigmask;
pthread_t *thread_ids = NULL;
pthread_mutex_t count_mtx = PTHREAD_MUTEX_INITIALIZER;
int thread_count;
pthread_t sig_thread;
pthread_mutex_t all_threads_done_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t all_threads_done_cond = PTHREAD_COND_INITIALIZER;
int all_threads_done = 0;
pthread_mutex_t Quit_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t Quit_cond = PTHREAD_COND_INITIALIZER;
int Quit = 0;

int
main(int argc, char **argv)
{
    PlayerArgs pa;

    if (init(argc, argv, &pa) < 0)
    {
        cerr << "Fatal initialization error\n";
        exit(-1);
    }
    if ((nsessions = register_sessions(smgrs, pa.nsessions))
        != pa.nsessions)
    {
        // currently should be fatal because it's easier
        cerr << "Couldn't join all sessions--fatal error\n";
        cleanup(-1);
    }
    int signo = 0,
        ret,
        i;
    if (sigemptyset(&player_sigmask) < 0)
    {
        perror("sigemptyset()");
        cleanup(0);
    }
    if (sigaddset(&player_sigmask, SIGINT) < 0)
```

```

{
    perror("sigaddset()");
    cleanup(0);
}
if (sigaddset(&player_sigmask, SIGUSR1) < 0)
{
    perror("sigaddset()");
    cleanup(0);
}
if (ret = pthread_sigmask(SIG_BLOCK, &player_sigmask, NULL))
{
    cerr << "pthread_sigmask() error: " << strerror(ret)
        << endl;
    cleanup(-1);
}
// Start signal handler thread
if (ret = pthread_create(&sig_thread, NULL,
                        handle_signals, &signo))
{
    cerr << "pthread_create(signal thread): "
        << strerror(ret) << endl << flush;
    exit(1);
}
while (pa.nrepeats)
{
    // debugging
    cout << "Starting threads...\n" << flush;
    /*
     * The "thread_count" variable is a misnomer, since
     * it doesn't keep track of how many threads there
     * are, but ONLY keeps track of how many *session*
     * threads there are (the main thread is NOT
     * considered a session thread).
     */
    set_thread_count(nsessions);
    /*
     * XXX Possible race condition: SIGINT received
     * prior to starting threads (or while starting
     * threads), hence before the condition check below.
     * pthread_signal() will be missed entirely.
     *
     * Prevent by locking all_threads_done_mtx first
     * thing in the while() loop?
     */
    if (start_threads(smgrs, &thread_ids, nsessions) < 0)
    {
        // currently should be fatal because it's easier
        cerr << "Couldn't start threads--fatal error\n";
    }
}

```

```

        cleanup(-1);
    }
    int r;
    if (r = pthread_mutex_lock(&all_threads_done_mtx))
    {
        cerr << "pthread_mutex_lock(): " << strerror(r) << endl <<
flush;
        exit(1);
    }
    if (r = pthread_cond_wait(&all_threads_done_cond,
                             &all_threads_done_mtx))
    {
        cerr << "pthread_cond_wait(): " << strerror(r) << endl <<
flush;
        exit(1);
    }
    if (all_threads_done)
    {
        if (r = pthread_mutex_unlock(&all_threads_done_mtx))
        {
            cerr << "pthread_mutex_unlock(): " << strerror(r)
                << endl << flush;
            exit(1);
        }
        I_am_done();
        // Wait for child threads
        for (i = 0; i < nsessions; ++i)
        {
            // debugging
            cout << "Joining thread " << i << "...\\n" << flush;
            if (ret = pthread_join(thread_ids[i], NULL))
            {
                cerr << "pthread_join(): " << strerror(ret) <<
endl
                << flush;
                exit(-3);
            }
            // close_session(smgrs[i]);
            cout << flush;
        }
        // don't exit here
    }
    else // uh oh
    {
        if (r = pthread_mutex_unlock(&all_threads_done_mtx))
        {
            cerr << "pthread_mutex_unlock(): " << strerror(r)
                << endl << flush;

```

```

        exit(1);
    }
    cerr << "Got a signal; abnormal exit...\n" << flush;
    I_am_done();
    for (i = 0; i < nsessions; ++i)
    {
        (void)pthread_join(thread_ids[i], NULL);
        close_session(smgrs[i]);
    }
    exit(-1);
} // end if (signo == SIGINT || signo == SIGUSR1)/else
// Make sure all sessions have same start time
struct timeval tv;
(void)gettimeofday(&tv, NULL);
for (i = 0; i < nsessions; ++i)
{
    smgrs[i]->dpss_poc->seek_dpss(0, SEEK_SET);
    if (pa.d)
        cout << "Resetting RTPComm...\n" << flush;
    smgrs[i]->reset();
    smgrs[i]->comm->set_start_time(tv);
}
free(thread_ids);
if (r = pthread_mutex_lock(&all_threads_done_mtx))
{
    cerr << "pthread_mutex_lock():" << strerror(r) << endl <<
flush;
    exit(1);
}
all_threads_done = 0;
if (r = pthread_mutex_unlock(&all_threads_done_mtx))
{
    cerr << "pthread_mutex_unlock():" << strerror(r) << endl
<< flush;
    exit(1);
}
if (pa.nrepeats > 0)
    --pa.nrepeats;
if (pa.d)
{
    if (pa.nrepeats > 1)
        cout << "Playing " << pa.nrepeats << " more times\n"
<< flush;
    else if (pa.nrepeats == 1)
        cout << "Playing once more\n" << flush;
    else if (pa.nrepeats < 0) // playing forever
        cout << "Playing again\n" << flush;
}

```

```

/*
 * Finally: if we were interrupted by SIGINT,
 * don't repeat--just go away
 */
if (r = pthread_mutex_lock(&Quit_mtx))
{
    cerr << ": pthread_mutex_lock(Quit):" << strerror(r)
        << endl << flush;
    exit(1);
}
if (Quit)
    pa.nrepeats = 0;
if (r = pthread_mutex_unlock(&Quit_mtx))
{
    cerr << ": pthread_mutex_unlock(Quit):" << strerror(r)
        << endl << flush;
    exit(1);
}
}
for (i = 0; i < nsessions; ++i)
{
    close_session(smgrs[i]);
}
if (!(pa.flags & PLAYER_QUIET))
{
    cout << "Done playing\n";
}
cleanup(0);
}
/* end main */

/*****
char *
make_toolname(void)
{
    char *basetoolname = "DMRP player";
    char *vers = ", v. 0.3";

    char *toolname = new char[strlen(basetoolname)+strlen(vers)+1];
    strcpy(toolname, basetoolname);
    strcat(toolname, vers);
    return(toolname);
}
/* end make_toolname(void) */

*****/

```

[illegible]


```

(int)pa_p->d);
}
catch (SessionMgrExc e)
{
    e.print();
    return(-1);
}
/*
 * XXX  A hack to make sure functionality ripped
 * out of here and dumped into the SessionManager
 * can print or suppress printing in the same way
 */
smgrs[i]->flags = pa_p->flags;
if (pa_p->sessions[i].set_id >= 0)
{
    if (smgrs[i]->open_dpss(pa_p->sessions[i].set_id,
                           pa_p->dpss_host,
                           pa_p->dsm_host) < 0)
    {
        cerr << "Error creating DPSSPOC " << i << endl;
        return(-1);
    }
}
else if (pa_p->sessions[i].filename)
{
    if (smgrs[i]->open_dpss(pa_p->sessions[i].filename,
                           pa_p->dpss_host,
                           pa_p->dsm_host) < 0)
    {
        cerr << "Error creating DPSSPOC " << i << endl;
        return(-1);
    }
}
else
{
    cerr << "No valid DPSS filename or set ID found for
session "
        << i << endl;
    return(-1);
}
} // end for()
delete[] toolname;
return(0);
}
/* end init(int, char **, PlayerArgs *) */

```

```

/*****

```

```

/*
 * Required because Solaris native compiler will not allow
 * pthread_create() to call SessionManager::mcast_send_thr()
 * directly for some reason.
 */
void *
run_session(void *targs)
{
    char *func = "run_session(void *)";
    SessionManager *sm_p = (SessionManager *)targs;
    if (sm_p->mcast_send() < 0)
    {
        cerr << "Error sending data!\n" << endl;
    }
    if (dec_thread_count() <= 0)
    {
        int r;
        if (r = pthread_mutex_lock(&all_threads_done_mtx))
        {
            cerr << "pthread_mutex_lock():" << strerror(r) << endl <<
flush;
            exit(1);
        }
        if (!all_threads_done)      // if all_threads_done, we got a
SIGINT
        {
            all_threads_done = 1;
            cout << "\tall threads done\n" << flush;
            if (r = pthread_cond_signal(&all_threads_done_cond))
            {
                cerr << "pthread_cond_signal():" << strerror(r)
<< endl << flush;
                exit(1);
            }
        }
        if (r = pthread_mutex_unlock(&all_threads_done_mtx))
        {
            cerr << "pthread_mutex_unlock():" << strerror(r) << endl
<< flush;
            exit(1);
        }
    }
    return((void *)0);
}
// end run_session(void *)

/*****/

```

```

// Exactly the same as start_threads() in listener.cc
int
start_threads(SessionManager **smarray, pthread_t **tids_pp, int ns)
{
    char *func = "start_threads(SessionManager **, pthread_t **,
int)";
    pthread_attr_t tattr;
    int err;

    if (err = pthread_attr_init(&tattr))
    {
        cerr << func << ": pthread_attr_init() error (" <<
strerror(err)
        << ")\n" << flush;
        return(-1);
    }
    if (err = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM))
    {
        cerr << func << ": pthread_attr_setscope() error (" <<
strerror(err)
        << ")\n" << flush;
        return(-1);
    }
    *tids_pp = (pthread_t *)calloc(ns, (int)sizeof(pthread_t));
    if (*tids_pp == NULL)
    {
        cerr << "Error creating thread ID array; out of memory?\n" <<
flush;
        return(-1);
    }
#ifdef SOLARIS
    thr_setconcurrency(ns+1);      // +1 for signal-watching thread
#endif
    for (int i=0; i < ns; ++i)
    {
        int r = pthread_create((*tids_pp)+i, &tattr,
                                run_session, smarray[i]);
        if (r)
        {
            cerr << "Error creating thread for session " << i << endl
<< flush;
            // continue?
            return(-1);
        }
    }
    return(0);
}
// end start_threads(SessionManager **, pthread_t **, int)

```

```

/*****/
void *
handle_signals(void *targs)
{
    char *func = "handle_signals(void *)";
    int *signo = (int *)targs,
        ret;

    if ((ret = sigwait(&player_sigmask, signo)) < 0)
    {
        cerr << func << ": " << strerror(ret) << endl << flush;
        exit(-2);
    }
    if (*signo == SIGINT)
    {
        int r;
        // make sure main() knows not to loop for a replay
        if (r = pthread_mutex_lock(&Quit_mtx))
        {
            cerr << func << ": pthread_mutex_lock(Quit):" <<
strerror(r)
                << endl << flush;
            exit(1);
        }
        Quit = 1;
        if (r = pthread_mutex_unlock(&Quit_mtx))
        {
            cerr << func << ": pthread_mutex_unlock(Quit):" <<
strerror(r)
                << endl << flush;
            exit(1);
        }
        if (r = pthread_mutex_lock(&all_threads_done_mtx))
        {
            cerr << func << ": pthread_mutex_lock():" << strerror(r)
                << endl << flush;
            exit(1);
        }
        if (!all_threads_done)
        {
            all_threads_done = 1;
            cout << func << ": all threads done\n" << flush;
            if (r = pthread_cond_signal(&all_threads_done_cond))
            {
                cerr << func << ": pthread_cond_signal():" <<
strerror(r)

```

```

        << endl << flush;
        exit(1);
    }
}
if (r = pthread_mutex_unlock(&all_threads_done_mtx))
{
    cerr << func << ": pthread_mutex_unlock():" <<
strerror(r)
        << endl << flush;
        exit(1);
    }
}
else
{
    cerr << func << ": received signal " << *signo << endl <<
flush;
    exit(1);
}
return((void *)0);
}
// end handle_signals(void *)

/*****
void
#ifdef OS_SOLARIS_2_4
cleanup(int sig)
#else
cleanup(...)
#endif
{
    if (!nsessions)
        cerr << "No sessions!\n";
    return;
}
*/ end cleanup */

/*****
int
register_sessions(SessionManager **sessions, int ns)
{
    char *func = "register_sessions()";
    int nsuccessful = 0;

    struct timeval tv;
    (void)gettimeofday(&tv, NULL);

```

```

for (int i = 0; i < ns; ++i)
{
    // address and port initialized by memcpy() in init()
    if (sessions[i]->register_session() < 0)
    {
        cerr << "Couldn't register session at address "
              << sessions[i]->si.addr << "/"
              << sessions[i]->si.port << endl;
        break;
    }
    else
    {
        // Make sure all sessions have same start time
        sessions[i]->comm->set_start_time(tv);
        /* Make sure sessions advertise themselves
           as "replay," initially */
        sessions[i]->comm->set_name("DMRP replay");
        ++nsuccessful;
    }
}
return(nsuccessful);
}
// end register_sessions(SessionManager **, int)

/*****
void
set_sig_handlers(void)
{
    struct sigaction action;

    memset(&action, 0, (int)sizeof(action));
#ifdef SOL2_5_SIGNALS
    action.sa_handler = cleanup;
#else
    action.sa_handler = (void(*)())cleanup;
#endif
    if (sigaction(SIGINT, &action, NULL) == -1)
    {
        perror("sigaction");
        cerr << "Unable to set SIGINT handler\n";
        exit(-1);
    }
    return;
}
*/
end set_sig_handlers(void) */

```

```

/*****/
void
close_session(SessionManager *sm_p)
{
    if (sm_p->dpss_poc->close_dpss() < 0)
    {
        cerr << "Error closing DPSS/DSM connections; continuing
cleanup\n"
                << flush;
    }
    sm_p->leave_session();
    return;
}
// end close_session(SessionManager *)

/*****/
void
I_am_done(void)
{
    char *func = "I_am_done(void)";

    for (int i=0; i < nsessions; ++i)
    {
        smgrs[i]->quit();
    }
    return;
}
// end I_am_done(void)

/*****/
void
set_thread_count(int nthreads)
{
    mrp_lock_mutex(&count_mtx);
    thread_count = nthreads;
    mrp_unlock_mutex(&count_mtx);
    return;
}
// end set_thread_count(int)

/*****/
int
dec_thread_count(void)
{
    mrp_lock_mutex(&count_mtx);

```

```

    int tc = --thread_count;
    mrp_unlock_mutex(&count_mtx);
    return(tc);
}
// end dec_thread_count(void)

```

B.4.3 *player_utils.cc*

```

// Need to be declared globally to prevent compiler from complaining
extern char *optarg;
extern int optind;

/*****/
void
init_PlayerArgs(PlayerArgs *pa_p)
{
    if (pa_p == NULL)
        return;
    memset(pa_p, 0, sizeof(PlayerArgs));
    pa_p->flags = 0;
    pa_p->ttl = DMRP_DEFAULT_TTL;    // defined in MRPCCommon.h
    pa_p->nrepeats = 1;    // play back file once
    pa_p->d = 0;
    pa_p->v = 0;
    pa_p->sessions =
        (DMRPSessionInfo *)calloc(1, sizeof(DMRPSessionInfo));
    if (pa_p->sessions == NULL)
    {
        cerr << "Couldn't allocate DMRPSessionInfo; out of
memory?\n";
        exit(1);
    }
    pa_p->nsessions = 1;
    for (int i = 0; i < pa_p->nsessions; ++i)
    {
        pa_p->sessions[i].set_id = -1;
    }
    return;
}
/* end init_PlayerArgs(PlayerArgs *) */

/*****/
void
parse_args(int ac, char **av, PlayerArgs *pa_p)
{
    // x only for development

```



```

const char *PLAY_ARGS = "F:b:d:f:hi:m:qr:s:t:vx";
const char *opts = PLAY_ARGS;
char *prog = av[0];
int c;

while ((c = getopt(ac, av, opts)) != EOF)
{
    switch (c)
    {
        case 'F':
            if (optarg == NULL)
                usage(prog);
            pa_p->sesfile = optarg;
            break;
        case 'b':
            if (optarg == NULL)
                usage(prog);
            pa_p->bandwidth = (ui32)atoi(optarg);
            break;
        case 'd':
            if (optarg == NULL)
                usage(prog);
            pa_p->d = (ui16)atoi(optarg);
            break;
        case 'f':
            if (optarg == NULL)
                usage(prog);
            pa_p->sessions[0].filename = optarg;
            break;
        case 'h':
            usage(prog);
            break;
        case 'i':
            if (optarg == NULL)
                usage(prog);
            pa_p->dpss_host = optarg;
            break;
        case 'm':
            if (optarg == NULL)
                usage(prog);
            pa_p->dsm_host = optarg;
            break;
        case 'q':
            pa_p->flags |= PLAYER_QUIET;
            break;
        case 'r':
            if (optarg == NULL)
                usage(prog);

```

```

        pa_p->nrepeats = atoi(optarg);
        break;
    case 's':
        if (optarg == NULL)
            usage(prog);
        pa_p->sessions[0].set_id = atoi(optarg);
        break;
    case 't':
        if (optarg == NULL)
            usage(prog);
        pa_p->ttl = atoi(optarg);
        break;
    case 'v':
        pa_p->v = 1;
        break;
    case 'x':
        pa_p->flags |= Experimental;
        break;
    case '?':
    default:
        usage(prog);
    }
}
if (pa_p->sesfile)
{
    free(pa_p->sessions);
    pa_p->sessions = NULL;
    // debugging
    cout << "Reading session info...\n";
    pa_p->nsessions = read_conference_file(pa_p->sesfile,
&pa_p->sessions);
    if (pa_p->nsessions <= 0)
    {
        cerr << "Error parsing conference configuration file!\n";
        exit(-2);
    }
    for (int i = 0; i < pa_p->nsessions; ++i)
    {
        pa_p->sessions[i].ttl = pa_p->ttl;
        pa_p->sessions[i].bandwidth = pa_p->bandwidth;
        if (pa_p->d)
            print_DMRPSessionInfo(pa_p->sessions+i);
    }
}
else if (optind < ac && get_addr_info(av[optind],
&pa_p->sessions[0].addr,
&pa_p->sessions[0].port)
    < 0)

```

